jamk.fi

NESTBOT – KESKUSTELEVA TEKOÄLYKAVERI

Arttu Maijanen

Opinnäytetyö
Toukokuu 2014

Mediatekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

KUVAILULEHTI

Tekijä(t) MAIJANEN, Arttu	Julkaisun laji Opinnäytetyö	Päivämäärä 12.05.2014
	Sivumäärä 65	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty (X)

Työn nimi
NESTBOT – KESKUSTELEVA TEKOÄLYKAVERI

Koulutusohjelma
Mediatekniikan koulutusohjelma

Työn ohjaaja(t)
MANNINEN, Pasi

Toimeksiantaja(t)
SkyNEST-projekti
RINTAMÄKI, Marko

Tiivistelmä
Opinnäytetyön päämääränä oli luoda FreeNest-projektialustaa varten suunniteltu tekoälyhahmo, jonka kanssa järjestelmän käyttäjät voivat käydä tekstimuotoisia keskusteluita. Tekoälyn luomista varten tutustuttiin tekoälytutkimuksen piirissä syntyneisiin teorioihin ja menetelmiin. Erityisen huomion kohteena olivat ns. chatterbotit, jotka vastaanottavat käyttäjältä tekstisyötettä, tulkitsevat sen sisältöä ja antavat käyttäjälle jonkin aiheeseen liittyvän vastauksen.

Uutta tekoälyä ryhdyttiin toteuttamaan JavaScript-pohjaisena web-sovelluksena. Pääasialliset kehityshaasteet liittyivät tekoälyn lauseenkäsittelyalgoritmeihin ja sanastorakenteeseen. Lopullinen tekoäly kykeni tunnistamaan käyttäjän antamasta syötteestä avainsanoja, joiden perusteella se pystyi antamaan aiheeseen liittyvän vastauksen. Tekoäly pystyi myös toistamaan omissa vastauksissaan otteita käyttäjän antamasta syötteestä.

Tekoälyhahmoa edustamaan luotiin myös animaatiohahmo, joka toteutettiin JavaScriptillä HTML5:n canvas-elementille. Grafiikkaohjelmoinnin helpottamiseksi ja sovelluksen suorituskyvyn parantamiseksi opinnäytetyössä perehdyttiin myös vapaasti saatavilla oleviin JavaScript-grafiikkakirjastoihin. Opinnäytetyössä vertailtiin toisiinsa Processing.js-, Kinetic.js- sekä EaselJS-grafiikkakirjastoja, joista lopulliseen sovellukseen valittiin käytettäväksi EaselJS.

Lopullisessa sovelluksessa yhdistettiin lauseenkäsittely ja grafiikka virtuaalihahmoksi, joka vastausten antamisen lisäksi reagoi käyttäjän tekstisyötteeseen ilmeillä ja eleillä. Sovellukseen sisällytettiin myös mahdollisuus kokonaan uuden hahmografiikan luomiseen ja käyttämiseen.

Avainsanat (asiasanat)
Tekoäly, lauseenkäsittely, hahmoanimaatio, HTML5 canvas, JavaScript

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s)	Type of publication	Date
MAIJANEN, Arttu	Bachelor´s Thesis	12.05.2014
	Pages 65	Language Finnish
		Permission for web publication (X)

Title
NESTBOT – AI FRIEND FOR CASUAL CONVERSATION

Degree Programme
Media Engineering

Tutor(s)
MANNINEN, Pasi

Assigned by
SkyNEST Project
RINTAMÄKI, Marko

Abstract
The purpose of the thesis was to create an AI character for the FreeNest project platform and enable it to have text conversations with FreeNest users. The theory and methods of AI research were studied to aid the creation of the character. Of particular interest were so-called chatterbots, i.e. artificial intelligences that receive text input from the user, process its content and form a coherent response based on it.

The new AI was built as a JavaScript-based web application. The primary development challenges involved the AI's sentence processing algorithms and vocabulary structure. The final AI could recognize keywords within the user's input and produce relevant responses based on them. The AI could also repeat parts of the user's input within its own responses.

A graphical representation in the form of an animated character was also created for the AI. The character was created using JavaScript and the HTML5 canvas element. Freely available JavaScript graphics libraries were studied in the hope of aiding graphics programming and increasing application performance. The libraries compared in the thesis were Processing.js, KineticJS and EaselJS, of which EaselJS was chosen as the library to be used in the final application.

The final application combined sentence processing and character animation into a virtual character that can react to the user's input with expressions and gestures. The application also includes the possibility to create and use entirely new graphical representations for the character.

Keywords
Artificial intelligence, sentence processing, character animation, HTML5 canvas, JavaScript

Miscellaneous

Sisältö

Kuviot

Taulukot

1 Työn lähtökohdat

1.1 Opinnäytetyön tausta

Kesällä 2013 Jyväskylän ammattikorkeakoulun omistaman SkyNest-projekti järjesti Summer Factory 2013 -työharjoittelun. SkyNest-projektin pääasiallinen tuote on FreeNest-projektialusta, johon on koottu sekä avoimesti saatavilla olevia että erikseen sitä varten kehitettyjä web-pohjaisia työkaluja. Harjoittelun aikana heräsi ajatus ”projektieläimestä”, eli jonkinlaisesta FreeNest-järjestelmään kuuluvasta virtuaalisesta maskottihahmosta, jonka kanssa käyttäjä voisi keskustella. Virtuaalihahmon ei olisi tarkoitus olla konkreettisesti projektinhallintaa helpottava työkalu, vaan pikemminkin omalaatuinen lisämauste, jota ei löydy muista vastaavista järjestelmistä ja joka auttaisi FreeNestiä erottumaan markkinoilla. Kukaan ei varsinaisesti kuitenkaan ollut suunnittelemassa tai kehittämässä tällaista projektieläintä.

Projektieläimen konseptin pohjalta ruvettiin kuitenkin tekemään opinnäytetyötä muutama kuukausi harjoittelun päättymisen jälkeen. Aihe valittiin, koska se yhdistelee mielenkiintoisesti grafiikkaa, web-sovelluksia ja tekoälyohjelmointia. Se on myös kunnianhimoinen ja sen verran lystikäskin, että se toimi käytännönläheisenä johdatuksena tekoälytutkimuksen aihepiiriin.

1.2 Tavoitteet ja menetelmät

Tekoälytutkimus on monipuolinen ja laaja tieteenala, jonka sovellutukset arkipäiväistyvät jatkuvasti. Esimerkiksi nettikauppojen antamien automaattisten tuotesuositusten taustalla toimii usein tekoälyalgoritmi, joka jäsentelee eri käyttäjien ostokäyttäytymisiä yhtenäisiksi asiakasprofiileiksi. Tällaisten yhtä tehtävää suorittavien “robottien” lisäksi tekoälytutkimus käsittelee myös mahtipontisempia aiheita, kuten oppivien tekoälyjen ohjelmointia tai ihmisen tietoisuuden mallintamista. Opinnäytetyön tavoitteena oli perehtyä tekoälytutkimukseen mahdollisimman laaja-alaisesti, jotta projektieläimen toteutukseen saataisiin käyttöön hyviä menetelmiä ja vankka teoreettinen pohja.

Käyttäjän kanssa keskusteleva tekoäly ei ole ajatuksena uusi, ja monet olemassa olevista tekoälyistä ovat huomattavasti sofistikoituneempia kuin alemman insinöörityt-kinnon yhteydessä olisi edes järkevää yrittää tehdä. Opinnäytetyössä pyrittiin kuitenkin rakentamaan yksinkertaisen versio tällaisesta tekoälystä, jotta monimutkaiseen aihepiiriin pystyttäisiin tutustumaan selkeiden käytännön tavoitteiden kautta. Tavoitteena oli myös luoda näyttävä ja omaperäinen sovellus, joka kiinnittää huomiota ja toimii osana FreeNestin markkinointia.

Opinnäytetyöhön liittyvä sovellus toteutettiin web-sovelluksena, joka sisälsi HTML5:n canvas-elementille piirrettävää grafiikkaa ja animaatioita. Tehtävän helpottamiseksi opinnäytetyössä tutustuttiin JavaScriptin grafiikkakirjastoihin, joista muutama otettiin tarkempaan kokeiluun ja vertailuun opinnäytetyön tarkoituksiin sopivimman kirjaston löytämiseksi.

2 Chatterbot - keskusteleva tekoäly

2.1 Mitä on tekoäly?

Tekoälyn käsite herättää usein mielikuvia tieteiselokuvien tietokoneista ja roboteista, jotka keskustelevat ihmisten kanssa ja ovat älykkyydeltään ainakin heidän vertaisiaan, usein myös selvästi älykkäämpiä. Eräs termi tällaiselle ihmistason saavuttaneelle tai sen rajat ylittäneelle tekoälylle on vahva tekoäly. Toistaiseksi vahvoja tekoälyjä esiintyy vain tieteistarinoissa, ja nykyhetken tekoälyjä nimitetään vastaavasti heikoiksi tekoälyiksi: niillä on jokin rajattu toiminta-alue, jolla ne saattavat toimia selkeästi ihmistä tehokkaamminkin, mutta mistään yleisestä älykkyydestä niiden kohdalla ei voida puhua. (Viitaniemi 2008, 49.)

Deep Blue saattoi esimerkiksi voittaa Kasparovin shakissa, mutta sama kone ei olisi kyennyt ennustamaan pörssikursseja tai suunnittelemaan taloja. Samankaltainenkin toiminta, kuten esimerkiksi tammen pelaaminen, olisi varmasti vaatinut sen tekoälyltä merkittävää uudelleenohjelmointia.

Nykyajan arkipäiväiset tekoälysovellukset ovat usein yllättävän huomaamattomia taustaprosesseja, jotka eivät välttämättä herätä heti mielikuvia elokuvien robottiapureista tai tietokoneaivoista. Ne ovat kuitenkin vahvasti erikoistuneita järjestelmiä, joiden vastuulla voi olla tärkeitä tehtäviä kuten robottikirurgia, luottopäätösten teko tai tietokoneiden suunnittelu. (Mts. 49.)

Hyvänä esimerkkinä huomaamattomasta mutta tärkeästä tekoälysovelluksesta on Googlen hakualgoritmi, joka pyrkii yhdistelemään relevanttia tietoa oikeisiin hakusanoihin. Yksi Googlen pitkän tähtäimen tavoitteista onkin luoda yhä vahvempi tekoäly, joka ymmärtäisi ja mallintaisi ihmisen ajattelutapaa paremmin. (Mts. 90–91.)

Hieman kevyempiä ovat erilaisiin peleihin tai virtuaalihahmoille suunnitellut tekoälyt, joille riittää yleensä jo älykkään toiminnan vaikutelman luominen. “Projektieläimen” toimeksiannossa kyse oli nimenomaan virtuaalihahmosta, jonka kanssa käyttäjä voi keskustella. Käytännössä tämä tarkoittaa, että tekoäly tulkitsee käyttäjältä saamaan-

sa tekstisyötettä ja antaa sen perusteella käyttäjälle keskustelunomaisia vastauksia. Tällaisesta tekoälystä käytetään muun muassa yleisnimitystä chatterbot, ja niiden tutkimuksen ja kehityksen historia ulottuu tekoälytutkimuksen alkuajoille.

2.2 Inhimillinen äly

Jos virtuaalihahmon tekoälyn on tarkoitus toimia hieman ihmisälyn tavoin, lienee syytä perehtyä vielä tarkemmin inhimillisen älykkyyden käsitteeseen. Älykkyyden tarkat määritelmät ovat kuitenkin olleet kiistanalaisia. On jopa esitetty, ettei mitään yleismaailmallista älykkyyttä ole olemassa, vaan kyseessä on parhaimmillaankin vain nimitys sille suureelle, joita älykkyystesteissä mitataan. Toiset määrittelevät älykkyyden laajemmin yleisenä oppimis-, päättely- ja mukautumiskykynä. (Smith, Nolen-Hoeksema, Fredrickson, Loftus 2003, 427.)

Jotkin älykkyyttä määrittelevät teoriat ovat jakaneet älykkyyden erillisiin osa-alueisiin, kun taas toiset tiivistävät sen yhdeksi yleiseksi tiedonkäsittelyprosessiksi. Älykkyyden perustaksi on esitetty sekä biologisia tekijöitä että sosiaalista toimintaa. Eri kulttuurien näkemykset älykkyydestä ovat myös vaikuttaneet teorioihin. (Mts. 436–443.)

Inhimillistä älykkyyttä on kuitenkin lopulta hankalaa tiivistää yksittäiseksi mitattavaksi suureeksi, joten yleensä onkin tyydytty määrittelemään erilaisia älykkään toiminnan osa-alueita. (Inhimillinen kone, konemainen Ihminen 2001, 18.)

Viitaniemi (2008, 39–41) jakaa ajattelun karkeasti vaistonvaraiseen toimintaan, aiemmin opitun toiminnan refleksinomaiseen toistoon ja lopulta tietoiseen toimintaan. Vasta tietoinen toiminta edellyttää välttämättömästi ympäristön ja ärsykkeiden havainnointia. Vertaamme ärsykkeitä aiemmin keräämäämme tietopohjaan, jonka perusteella valitsemme jonkin aiemman toimintamallin jota voimme käyttää joko sellaisenaan tai tilanteeseen mukautettuna. Oppimis- ja mukautumiskyky onkin tyypillinen yhdistävä tekijä erilaisille älykkyyden teorioille ja määritelmille.

2.3 Viisas kone

2.3.1 Viisaus – Hyviä ratkaisuja ja sosiaalista vuorovaikutusta

Eräs älykkyystutkimuksessa ilmennyt älykkyyden osa-alue on viisaus. Lyhyesti kuvailtuna viisaus on jonkinlainen yleismaailmallinen ominaisuus, joka ilmenee hyvien ratkaisujen sarjana ja jota käytetään yhteisön hyödyksi. Viisauteen kuuluvat muun muassa ongelmien tunnistaminen, ratkaisujen muodostaminen ja lopputuloksista oppiminen. Viisaudelle on myös tunnusomaista se, että se on luonteeltaan sosiaalista: viisas yksilö on siis sellainen, joka tunnustetaan viisaaksi omassa yhteisössään. (Nurmi, Ahonen, Lyytinen, Lyytinen, Pulkkinen, Ruoppila 2006, 230–233.)

Tietokoneohjelman älykkyyttä arvioitaessa tällainen käytännönläheinen älykkyys tuntuisi järkevältä mittarilta. Jos yhteisö saadaan tunnustamaan ohjelman ratkaisut viisaiksi, sitä voitaisiin jo pitää aidosti älykkäänä. Nykyhetken tekoälyt ovat kuitenkin aiemmin kuvaillun kaltaisia vaivihkaisia taustaprosesseja, joita emme välttämättä miellä sosiaalisiksi toimijoiksi samalla tavoin kuin elokuvien robottiapureita. Näin ollen emme yleensä ryhdy ylistämään Googlen viisautta, vaikka se saattaisikin johdattaa meidät yhdellä hakusanalla juuri oikean tiedon äärelle.

2.3.2 Tekoäly sosiaalisena toimijana

On kuitenkin olemassa jonkin verran näyttöä siitä, että ihminen noudattaa tietokonetta käyttäessään samoja sosiaalisia sääntöjä kuin ihmisten välisessä vuorovaikutuksessakin. Tähän liittyviä ilmiöitä käsittelee affektiivinen tietojenkäsittely, eli tunteisiin liittyvä laskenta. Tutkimusalue kattaa muun muassa tunteita aistivien koneiden suunnittelun: kone voisi esimerkiksi tarkkailla ihmistä kameralla, havainnoida hänen kasvoistaan erilaisten tunnereaktioiden aiheuttamia fysiologisia vasteita ja säätää omaa toimintaansa sen mukaan. Vastaavasti on tutkittu myös ihmisen reaktioita koneen toimintaan, kuten esimerkiksi erilaisten virheilmoitusten käyttäjässä herättämiä tuntemuksia. Positiivisen tunnekokemuksen saaneet käyttäjät haluavat esimerkiksi toimia koneen kanssa pitempään. (Inhimillinen kone, konemainen ihminen 2001, 44–45.)

Vastaavasti ihmistenkin välisestä vuorovaikutuksesta on havaittu, että tunnereaktiot vaikuttavat vuorovaikutuksen jatkamiseen tai siitä pois hakeutumiseen. Tällaiset tunnereaktiot vaikuttavat myös ihmisen oppimiskykyyn, kommunikaatiokykyyn ja muistin toimintaan. (Mts. 37–38.)

Negatiivisia tunnereaktioita herättävä sovellus luonnollisesti siis huonontaa käyttäjän työtehokkuutta, joten sosiaalisesti älykäs sovellus johtaisi ainakin teoriassa tehokkaampaan työskentelyyn. Toki on syytä muistaa, että tekoäly voisi päätyä tässäkin asiassa ajoittain vääriin ratkaisuihin. Tuskin kukaan haluaisi esimerkiksi olla vuorovaikutuksessa sellaisen sosiaalisen koneen kanssa, joka vahingossa loukkaisi käyttäjänsä tunteita. Tällaista konetta emme varmastikaan myös haluaisi tunnustaa viisaaksi.

Lopulta viisauden tarkastelu ei ehkä edes ole sopiva tapa vertailla inhimillistä älyä tekoälyyn. Yleisesti pidämme kaikkia ihmisiä kuitenkin jollain tapaa älyllisinä olentoina, vaikka emme suoranaisesti nimittäisikään heitä viisaiksi. Yksimielisen tieteellisen kannan puuttuessa joudumme vain toteamaan, että ihmisen ja tekoälyn erottavana tekijänä lienee jonkinlainen selittämätön inhimillinen kipinä.

2.4 Kohti inhimillistä tekoälyä

2.4.1 Älykkyys yleisenä prosessina

Ihmisen kaltainen kone on kuitenkin ajatuksena kiehtova. Aihe herättää kysymyksiä siitä, miten ihminen itse asiassa ajattelee ja millaisia malleja sen kuvaamiseksi voidaan muodostaa. Voidaan jopa nostaa esille kysymyksiä siitä, onko koneen mahdollista saavuttaa jonkinlainen tietoisuus tai ajatella samalla tasolla kuin ihminen. Tällaisten aiheiden parissa työskentelee yleinen tekoälytiede, eli AGI (Artificial General Intelligence).

Yleinen tekoäly viittaa sellaiseen tekoälyyn, joka mallintaa ohjelmallisesti ihmisen ajattelutapaa. Yleisesti ottaen taustalla on ajatus siitä, että ihmisen tiedonkäsittelyssä on jokin perusprosessi, joka toimii sekä jokapäiväisen päättelyn että monipuolisen pohdinnan taustalla. Viitaniemi (2008, 50–51) korostaa tässä yhteydessä Occamin

periaatetta: yksinkertainen selitys on todennäköisemmin oikeassa kuin monimutkainen, joten tällaisen perusprosessin tulisi koostua minimaalisesta määrästä sääntöjä, jotka antaisivat kuitenkin maksimaalisen määrän tuloksia.

2.4.2 Koneellisen älykkyyden määrittely ja arviointi

Vuonna 1950 tekoälytutkija Alan Turing kehitti Turingin testin, jonka mukaan kone on älykäs jos sen kanssa keskusteleva ihminen ei kykene erottamaan sen vastauksia oikean ihmisen antamista vastauksista. Testi koostuu tekstimuotoisesta keskustelusta, jonka aikana arvioija esittää koneelle erilaisia kysymyksiä. Kuviossa 1 on esimerkki testitilanteesta, jossa arvioija yrittää päätellä, toimiiko hänen keskustelukumppaninaan oikea ihminen vai sellaista esittävä kone. Huomionarvoista testissä on se, että ainoastaan ohjelman tuottamat vastaukset ovat merkityksellisiä arvioinnin kannalta. (Inhimillinen kone, konemainen ihminen 2001, 9.)

Kuvio 1. Turingin testi.

Muun muassa Hutchens (1997, 19–20) onkin arvostellut testiä siitä, että sen kannalta on merkityksetöntä, onko ohjelma aidosti ajatteleva toimija vai ainoastaan suunniteltu huijaamaan testin arvioijia. Näin ollen voidaan pitää kyseenalaisena, onko testin

avulla mahdollista edes edistää tekoälytiedettä vai onko siihen osallistuminen parhaimmillaankin vain hupaisaa harrastustoimintaa.

Jotkin kognitiotieteilijät ovat puolestaan pohtineet, voitaisiinko testistä suoriutumisesta pitää aitona tietoisuuden merkkinä. Daniel Dennett ja Douglas Hofstadter väittävät, että jokainen testistä suoriutuva yksilö on väistämättä tietoinen. David Chalmers puolestaan väittää, että klassisen ajatuskokeen filosofinen zombi (täydellinen ihmiskopio joka käyttäytyy täysin inhimillisesti, mutta ei kuitenkaan omaa sisäisiä kokemuksia) voisi suoriutua testistä olematta siltikään tietoinen olento. Itse tietoisuuden luonteesta on myös lukuisia kilpailevia teorioita ja määritelmiä, joihin myös tekoälytutkimus on osaltaan vaikuttanut. (Viitaniemi 2008, 42–48.)

Turingin testiä suoritetaan muodollisesti vuosittain järjestettävässä Loebnerin kisassa, johon osallistuvat ohjelmat pyrkivät uskottelemaan arvioijille olevansa oikeita ihmisiä. Kisan pääpalkintoa ei tähän mennessä ole vielä voitettu, eli yksikään kisaan osallistunut ohjelma ei toistaiseksi ole onnistunut huijaamaan arvioijia. (Mts. 43.)

2.4.3 Tekoälyterapeutti ELIZA

Uskottavasti ajattelevan ohjelman luominen oli siis selkeästi liian korkealle asetettu tavoite opinnäytetyölle, mutta ajatus käyttäjän huijaamisesta tuntui mielenkiintoiselta näkökulmalta. Kuinka verrattain tyhmä ohjelma saataisiin harhauttamaan monipuolisesti ajattelevia ihmisiä?

Eräs klassinen esimerkki on Joseph Weizenbaumin vuonna 1966 julkaisema ELIZA-tekoäly, joka esittää psykoterapeuttia. ELIZA pääasiassa johdattelee käyttäjää kertomaan itsestään lisää ja toistelee varmistellen tämän lausuntoja. (Hutchens 1997, 4.)

ELIZAn huijauksesta tekee toimivan se, että ohjelman käyttökonteksti on rajattu juuri terapiatilanteeseen. Käyttäjä ei näin ollen ala kovin todennäköisesti keskustelemaan esim. säästä, urheilusta tai jostain muusta aiheesta, josta tekoälyllä ei ole taustatietoja. Tekoäly myös siirtää vastuun keskustelun eteenpäin viemisestä käyttäjälle pyytämällä tämän lausunnoista tarkennuksia tai esittämällä kysymyksiä jostain muusta

aiheesta. ELIZAn ympärillä liikkuukin paljon anekdootteja, joissa koekäyttäjät pitivät ohjelmaa aitona terapeuttina tai ainakin luotettavana vaihtoehtona sellaiselle.

Tällaisten uskomusten syntymistä nimitetään ns. ELIZA-efektiksi. Sen mukaan ihmiset ovat taipuvaisia uskomaan, että koneiden antamien viestien taustalla on jotain merkityksellisempiä toimintoja kuin vain tietyn tulosteen antaminen tietyssä tilanteessa. Toisin sanottuna käyttäjä uskoo ennemmin, että tekoäly on aidosti älykäs toimija sen sijasta, että se olisi suunniteltu vain vaikuttamaan sellaiselta. (Hofstadter 1995, 157–158.)

Opinnäytetyön kannalta oli rohkaisevaa huomata, miten suuri osa chatterbottien toiminnasta perustuu käyttäjän toiminnan ennakoimiseen ja johdatteluun. ELIZA olikin opinnäytetyötä varten tehdyn tekoälyn suurin yksittäinen esikuva, ja sen toiminnan tarkastelu antoi paljon innostusta ja hyvää lähdemateriaalia uuden tekoälyn luomista varten.

2.5 Tekoälyjen toimintamekanismit

2.5.1 Ongelmanratkaisua yksin ja yhteistyönä

Nykyhetken tekoälyt pyrkivät yleensä suorittamaan jotakin yksittäistä tehtävää, kuten esimerkiksi shakin pelaamista tai tiedon jäsentelyä. Tällöin tekoälyn toimintaa voidaan lähestyä puhtaana ongelmanratkaisuna. Mitä toimintoja tekoälyn on suoritettava, jotta tehtävä tulisi valmiiksi? Mitä pohjatietoja tekoälylle voidaan antaa tehtävän suoritusta varten?

Toisaalta puhtaasti ongelmanratkaisuun keskittyvä järjestelmä ei välttämättä edes ole erityisen älykäs. Se saattaa suorittaa mekaanisia laskutoimintoja hyvinkin tehokkaasti, mutta osoittamatta mitään luovuutta, oppimiskykyä tai muita merkkejä sellaisesta älykkyydestä, jota yleensä miellämme ihmisten käyttävän. Hofstadter (1995, 52–53) paneekin merkille, kuinka tekoälytutkimus on ikään kuin jakautunut kahtia ongelmanratkaisusta kiinnostuneisiin tutkijoihin ja abstraktimmista, älykkyyden luonteeseen liittyvistä kysymyksistä kiinnostuneisiin tutkijoihin. Näillä kahdella joukolla

on vähän tarjottavaa toisilleen - Hofstadter huomauttaa, että äärimmäisen sofistikoituneetkin shakkitekoälyt paljastavat ihmisten shakinpeluusta vain tapoja, joilla ihmiset eivät varmasti pelaa shakkia.

Tähänastiset esimerkit tekoälyistä ovat enimmäkseen olleet yksittäisiä ohjelmia, jotka työskentelevät itsenäisesti oman vastuualueensa parissa. Hajautettu tekoäly on tutkimusalue, jossa ongelmia pyritään ratkaisemaan usean erillisen tekoälyn eli agentin yhteistyönä. Agenteilla voi olla oma vastuualueensa tehtävän suorittamisessa, ja ne voivat myös välittää toisilleen käsittelemäänsä tietoa. Tästä seuraa myös yksi tutkimusalueen haasteista, sillä käytännössä tähän vaaditaan kykyä oppia muilta agenteilta saadun tiedon perusteella myös oman vastuualueen ulkopuolisia asioita. (Inhimillinen kone, konemainen ihminen 2001, 19–20.)

2.5.2 Oppivat tekoälyt

Oppimisesta puolestaan muodostuu omia haasteitaan tekoälyjen kehitykselle. Normaalisti tekoälylle on ohjelmoituna jonkinlaiset valmiit säännöt sen kohtaamia tilanteita varten. Shakkitekoälyllä saattaa esimerkiksi olla valmiina käytössään kehittäjien tallentamia listoja siirroista, joita kannattaa tehdä eri tilanteissa. Oppivalla shakkitekoälyllä sen sijaan olisi valmiina tietopohjanaan vain shakin perussäännöt, mutta se pystyisi lopulta oppimaan pelaamiensa pelien perusteella, millaiset siirrot ovat kannattavia.

Tekoälyjen oppimismenetelmät voidaan jakaa kolmeen luokkaan: ohjattu (supervised), vahvistettu (reinforced) ja ohjaamaton (unsupervised). Ohjatussa oppimisessa tekoälylle annetaan kunkin syötteen lisäksi lopputulos, johon tekoälyn halutaan päätyvän syötteen perusteella. Tekoäly siis etsii parhaansa mukaan tapoja, joilla määrättyyn lopputulokseen päästään, mutta ei poikkea siitä. Vahvistetussa oppimisessa tekoälyn annetaan itsenäisesti päätyä johonkin lopputulokseen, jonka toimivuuden ulkopuolinen arvioija arvioi. Ohjaamattomassa oppimisessa tekoäly saa tehdä omat päätöksensä saatujen tulosten toimivuudesta, mikä vaikuttaa tehottomalta mutta on toisaalta myös lähinnä ihmisen luontaista oppimis- ja havainnointiprosessia. (Inhimillinen kone, konemainen ihminen 2001, 20–21.)

Ihmisen oppimista on myös pyritty mallintamaan koneellisesti neuroverkoilla, eli ihmisen keskushermoston rakennetta simuloivilla tietorakenteilla. Oppiminen tapahtuu tällöin siten, että “hermosolujen” välille syntyy uusia kytköksiä ja olemassa olevat kytkökset vahvistuvat entisestään. Neuroverkot ovat luonnollisesti monipuolinen ja neurotieteeseen läheisesti liittyvä aihealue, ja niiden käytännön toteutukseen on useita menetelmiä. (Mts. 21–26.)

Hugo de Garis on tutkinut geneettisiä algoritmeja neuroverkoissa. Tässä yhteydessä geneettinen algoritmi viittaa siihen, että annetusta tehtävästä parhaiten suoriutuvan verkon “geenit” menevät jatkojalostukseen, jossa sen “jälkeläisestä” pyritään saamaan vielä tehokkaammin toimiva verkko. Järjestelmä oppisi näin ollen yksittäisten tietoyhteyksien muodostamisen lisäksi tehostamaan omaa havainnointi- ja oppimiskykyään. (Mts. 67–69.)

2.6 Kielen tunnistamisen ja tuottamisen haasteet

2.6.1 Orgaanisen kielen mekaaninen tulkinta

Luonnollisen kielen prosessointi on kielitiedettä ja tietojenkäsittelyä yhdistelevä tieteenhaara, joka käsittelee ihmiskielen koneellista tulkintaa. Tähän liittyy useita kapea-alaisempia tavoitteita, kuten esimerkiksi merkityksellisten sanojen erottaminen kieliopillisista rakenteista, sanojen tulkinta kontekstin perusteella ja erisnimien tunnistaminen. Tuhansien vuosien aikana orgaanisesti kehittyneet kielet eivät luonnollisesti noudata nykyaikaisen tietojenkäsittelyn logiikkaa, joten niiden tulkintaan liittyy huomattavia haasteita.

Esimerkiksi Loebnerin kisan keskustelulokeja lukemalla selviää, että testin arvioijat tekevät usein tahallaan kirjoitus- ja kielioppivirheitä tekoälyn hämäämiseksi. Ihminen yleensä tunnistaa pienet kirjoitusvirheet ja oivaltaa niiden oikeat merkitykset, kun taas niihin varautumaton tekoäly saattaa tulkita väärin kirjoitetun sanan kokonaan uudeksi sanaksi. Myös jotkin eri asioita ilmaisevat kieliopilliset rakenteet muistuttavat pintapuolisesti toisiaan, kuten esimerkiksi englannin kielen “your” ja “you’re”. Tällöin sinänsä oikein kirjoitettua sanaa saatetaan käyttää kieliopillisesti väärässä

yhteydessä, jolloin on ymmärrettävä kontekstin perusteella mitä lauseella halutaan ilmaista. Myös homonyymit, kuten esimerkiksi “mine” (kontekstista riippuen joko pronomini “minun”, substantiivi “kaivos”, tai verbi “louhia”) vaativat kielen tulkintaa syvällisemmin kuin yksittäisten sanojen tai rakenteiden merkitystasolla.

2.6.2 Joustava kielenkäyttö ja kontekstin huomiointi

Douglas Hofstadter (1995, 75–77) kuvailee joustavan kielenkäytön tulkinnan haasteita “me-too” -ilmiönä. Esimerkkinä hän antaa seuraavanlaisen sananvaihdon:

> *Shelley: I’m going to pay for my beer now.*
> *Tim: Me, too.*

Selvästikään Tim ei ole ryhtymässä maksamaan siitä samasta oluesta, jonka Shelley menee maksamaan. Semanttisella tasolla tulkittuna Timin lauseesta ei kuitenkaan tarkalleen ilmene, kenen oluesta Tim puhuu. “Minä myös” on itsenäisenä lauseena täysin abstrakti, ja sen tulkinta vaatii tilanteen kontekstin huomioimista. Hofstadter antaa myös seuraavanlaisen esimerkin:

> *Ana: My parents are always calling me “Lucie” and my sister “Ana”.*
> *Bob: Oh, yeah - my parents used to do that, too, except it was with our dog and cat.*

Robottimaisen looginen tulkinta tästä sananvaihdosta olisi, että Bobin vanhemmat nimittäisivät koiraansa ja kissaansa Lucieksi ja Anaksi. On kuitenkin epätodennäköistä, että Bobin vanhemmat olisivat antaneet lemmikeilleen täsmälleen samat nimet kuin Luciella ja hänen siskollaan on. Todennäköisempää on se, että Bobin vanhemmat vain sekoittavat lemmikkien tässä mainitsemattomat nimet toisiinsa. Tällainen tulkinta vaatii kuitenkin jo melko abstraktia päättelyä ja keskustelun implikaatioiden tulkintaa - kehittynytkin tekoäly saattaisi päätyä suoraviivaisempaan tulkintaan, jossa lemmikeillä sattuu olemaan samat nimet kuin sisaruksilla.

2.6.3 Ihmisen ja koneen erot kielen tulkinnassa

Hofstadter (1995, 93–95) on myös tutkinut paljon yksittäisten sanojen tunnistamismekanismeja. Toinen hänen kuvailemistaan kielen tunnistuksen haasteista on merki-

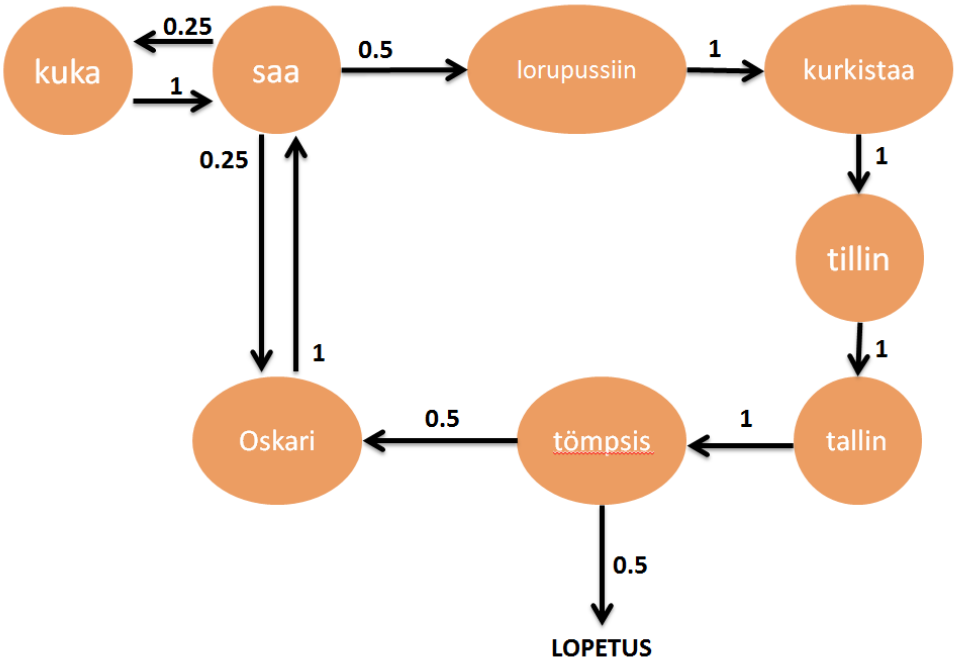
tykselisten sanojen rajakohtien erottaminen. Esimerkiksi sana “weeknights” merkitys voidaan tulkita oikeaoppisesti arki-illoiksi (“week” ja “nights”), mutta myös väärin pikkuruisiksi ritareiksi (“wee” ja “knights”). Hofstadter kiinnittää erityisesti huomiota siihen, kuinka tällaiset tulkinnat tapahtuvat äärimmäisen tarkalla kirjaimia, äänteitä ja semantiikkaa yhdistelevällä tasolla, mutta toisaalta myös enimmäkseen ilman tietoista prosessointia. Hofstadter huomauttaakin, että me ihmiset onnistumme välttämään merkittävän hyvin tällaisia virhetulkintoja niiden valtavasta määrästä huolimatta.

Kielen koneellisen prosessoinnin haasteet juontuvatkin usein siitä, että ihmisen tapa tulkita kieltä on erittäin tarkka ja täsmällinen prosessi, joka on kuitenkin niin abstrakti ja tiedostamaton, ettei sille voi määrittää selkeitä mekanismeja. Opinnäytetyön sovelluksen tuotannon aikana törmättiinkin moniin yllä kuvattuihin haasteisiin. Sofistikoituneet ratkaisut niihin olisivat käytännössä olleet jo kokonaan uuden opinnäytetyön veroinen aihe, joten tällaisten pulmien olemassaolo pyrittiin lähinnä tiedostamaan ja ohittamaan mahdollisimman hienovaraisesti. Jos tekoäly esimerkiksi vastaa monitulkintaiseen kysymykseen vaihtamalla kokonaan puheenaihetta, ei vielä suoranaisesti ilmene ettei se kykene tulkitsemaan sille annettua syötettä oikein. Tällainen menetelmä on lähinnä karu silmänkääntötemppu, mutta se ei varsinaisesti poikkea Loebnerin kisaankaan osallistuneiden tekoälyjen toimintatavoista.

2.6.4 Markovin ketjut – todennäköisyyksiin pohjautuva kielenkäyttö

Yllä annetut esimerkit ovat osoittaneet, kuinka olennaista tulkinnan kannalta on jonkinlainen käsitys keskustelun kontekstista ja itse keskustelutilanteesta. Kieltä voidaan kuitenkin ainakin tuottaa koneellisesti puhtaasti todennäköisyyksiin perustuvalla mallilla, jossa ohjelmalla ei edes huomioi mitä se on keskustelussa aiemmin sanonut. Muun muassa Jason Hutchens (1997, 13–14) hyödynsi omassa Loebnerin kisaan osallistuneessa tekoälyssään Markovin ketjua, joka on erilaisten tilojen välillä siirtymistä kuvaava todennäköisyysmalli. Lyhyesti selitettynä Markovin ketju koostuu erilaisista tiloista ja niiden välisistä yhteyksistä. Jokaisella tilojen välisellä yhteydellä on oma prosentuaalinen todennäköisyytensä. Tilat ja yhteydet muodostavat yhdessä todennäköisyyskartan, josta voidaan tarkkailla mihin tiloihin tarkasteltava järjestelmä voi

päätyä, millä todennäköisyyksillä ja minkä välitilojen kautta kulkien. Kuviossa 2 on esimerkki lastenlorun pohjalta luodusta Markovin ketjusta, jossa ympyrät edustavat ketjun eri tiloja. Nuolet puolestaan kuvaavat mahdollisia siirtymiä eri tilojen välillä, ja niiden vieressä olevat numerot edustavat kunkin siirtymän todennäköisyyttä.



Kuvio 2. Esimerkki Markovin ketjusta.

Yksi sovellus Markovin ketjuille on sattumanvaraisen mutta ainakin semanttisesti ymmärrettävän tekstin tuottaminen. Todennäköisyyskartta muodostetaan tässä tapauksessa jostakin laajasta tekstimateriaalista siten, että kartan jokainen tila vastaa tekstistä löytyviä sanoja. Sanojen välille luodaan yhteys, jos ne seuraavat lähdemateriaalissa toisiaan. Kun koko materiaali on käsitelty näin, saadaan malli siitä mitkä sanat kuuluvat todennäköisesti lauseessa peräkkäin. Mallin pohjalta voidaan nyt generoida uutta tekstiä valitsemalla kartalta jokin lähtötila ja siirtymällä aina satunnaisella todennäköisyydellä seuraavaan tilaan, ikään kuin heittämällä noppaa lautapeliä pelatessa. Riittävän suurella lähdemateriaalilla Markovin ketjut voivat tuottaa yllättävänkin koherentteja lauseita ilman varsinaista tiedon prosessointia tai mitään pohjalla olevaa kielitieteellistä mallia. Tässä tapauksessa tekoälystä puhuminen on kuitenkin jo kyseenalaista, ja lopputulos on kenties lähempänä aiemmin kuvailtua filosofista zombia.

Yksi esimerkki Markovin ketjuilla generoidusta tekstistä on Jyväskylän ammattikorkeakoulun tietorakenteet & algoritmit -opintojaksolle kirjoitettu Markovala-sovellus, joka toimi yllä kuvatulla tavalla ja käytti lähdetekstinään Kalevalaa. Se tuotti seuraavanlaisia, melko sekavia loruja ilman mitään ohjelmallista tietämystä suomen kielen rakenteista tai runoperinteistä:

> *Mieleni minun olisi, parempi olisi ollut merilohia, syvän aallon vastaeli: Viel' ei lie minulla aika näiltäpieniltä pesiltä, asunnoilta ahtahilta! Saata maalle kasvavalle, ahollen ylenevälle.*

Opinnäytetyön sovelluksessa päätettiin kuitenkin lopulta olla hyödyntämättä Markovin ketjuja niiden vaatiman lähdemateriaalin määrän vuoksi. Markovin ketjujen avulla on kuitenkin mahdollista tuottaa esimerkiksi chatterbotteja, jotka lisäävät käyttäjältä saadun syötteen aina osaksi omaa todennäköisyysmalliaan ja ikään kuin oppivat näin kielenkäyttöä käyttäjältä.

3 JavaScriptin grafiikkakirjastojen vertailu

3.1 HTML5:n canvasin ohjelmoimisen haasteet

Tekoälylle tarvittiin jokin graafinen esitys, joka päätettiin toteuttaa HTML5:n canvas-elementille piirrettynä animaatiohahmona. Canvas-elementin suora käsittely JavaScriptillä on kuitenkin jonkin verran mutkikkaampaa kuin esimerkiksi Flash-animaatioiden toteuttaminen ActionScriptillä. ActionScriptissä suuri osa matalan tason grafiikkarutiineista on automatisoitu, eikä ohjelmoijalle ole välttämätöntä työskennellä niiden parissa tai edes tuntea niiden toimintatapoja. Canvas sen sijaan vaatii ohjelmoijalta muun muassa ruudunpäivityksen hallintaa ja jonkin verran geometrian soveltamistakin. Opinnäytetyössä canvas-grafiikan suurimmiksi haasteiksi todettiin useamman graafisen elementin yhtäaikaisen käsittely sekä ruudunpäivityksen optimointi.

Canvas-animaatioissa useamman graafisen objektin käsittely vaatii jonkin verran ylimääräistä ajattelua verrattuna esimerkiksi juuri ActionScriptin oliomaiseen lähestymistapaan. Transformaatiot (kuten objektien siirtely, pyöritys ja skaalaus) vaikuttavat lähtökohtaisesti kerralla koko canvas-elementtiin ja kaikkeen sen sisältämään grafiikkaan. Näin ollen esimerkiksi yhden objektin pyöritys liikkumattoman taustan päällä vaatii hieman yksinkertaistetusti sekä objektin pyöritystä haluttuun suuntaan että taustan pyöritystä vastakkaiseen suuntaan. Näin syntyy vaikutelma siitä, että tausta pysyy paikallaan objektin pyöriessä sen päällä. Tämän ohjelmointi erikseen jokaista animaatiota varten johtaa melko pitkään ja epäselvään koodiin, joten joko omien grafiikkafunktioiden kirjoittaminen tai valmiiden grafiikkakirjastojen käyttäminen on suositeltavaa.

Canvasin ruudunpäivitys vaatii myös käytännössä omien ruudunpäivitysrutiinien kirjoittamista. Yksinkertaisin tapa olisi tyhjentää ja piirtää koko canvasin alue joka ruudunpäivityksellä uudelleen, mutta tämä on ymmärrettävästi hidasta ja tehotonta. Canvasin uudelleenpiirtäminen pienemmiltä nelikulmion muotoisilta alueilta on

mahdollista, mutta tämä vaatii luonnollisesti näiden alueiden oikeiden koordinaattien laskemista jokaista ruudunpäivitystä varten.

Voitaisiin siis sanoa, että canvas antaa ohjelmoijalle hyvin vapaat kädet grafiikan tarkan toiminnan suhteen. Kokemattoman ohjelmoijan käsissä tämä voi kuitenkin johtaa epäselvään ja huonosti toimivaan koodiin. Tämän vuoksi oli syytä tutustua hieman JavaScriptin grafiikkakirjastoihin, joilla saataisiin kirjoitettua siistimpää koodia ja käytettyä vähemmän aikaa suoritustehon optimointiin.

Lopulta päädyttiin valitsemaan muutama vapaasti lisensoitu kirjasto ja kokeilemaan, kuinka ne käytännössä sopivat opinnäytetyön tarkoituksiin. Testiin valitut kirjastot olivat Processing.js, KineticJS sekä EaselJS. Kukin kirjastoista sisältää samankaltaisen valikoiman animaatiometodeja, grafiikkasuodattimia ja muita valmiita ratkaisuja canvas-animaatioiden tekemiseen. Opinnäytetyön tarpeiden kannalta yhdessäkään kirjastossa ei ollut mitään ainutlaatuista ominaisuutta tai työkalua joka olisi puuttunut muista, joten käytettävän kirjaston valintaan tarvittiin tarkempaa vertailua.

3.2 Vertailumenetelmät

Jokaisella valitulla grafiikkakirjastolla toteutettiin samanlainen testianimaatio, jossa staattisen bittikarttataustan päällä on viisi jatkuvasti pyörivää, liikkuvaa ja skaalautuvaa bittikarttakuvaa. Kuviossa 3 on kuvankaappaus EaselJS:llä toteutetusta testianimaatiosta.

Kuvio 3. Grafiikkakirjastojen testianimaatio.

Tällainen testi tuntui vastaavan melko hyvin lopullisen sovelluksen vaatimuksia, ja se sisältäisi myös riittävän raskaita grafiikkaoperaatioita eri kirjastojen suoritustehon mittaamiseen. Valmiit animaatiot testattiin useilla laitteilla ja selaimilla, jotta kirjastojen tehokkuudesta saataisiin tarkempaa tilastotietoa ja mahdolliset yhteensopivuusongelmat tulisivat ilmi.

Jokainen testianimaatio sisälsi myös frames per second -mittarin, jotta suoritustehosta saataisiin välittömästi jonkinlainen lukema. Käytännössä tämä kuitenkin osoittautui huonoksi tavaksi saada vertailukelpoista tietoa eri kirjastojen tehokkuudesta, sillä täyteen 60 fps:n kuvataajuuden yltäneitä mittaustuloksia ei voinut enää asettaa paremmuusjärjestykseen. Eri kirjastot saattoivat myös esittää lukeman pidemmän ajanjakson keskiarvona, joka ei huomioinut hetkellisiä piikkejä tai tipahduksia. Mobiililaitteilla tai vanhemmilla koneilla fps-luku antoi kuitenkin selkeämpää näyttöä eri kirjastojen suoritustehoista.

Tarkempia lukuja saatiin Chromen kehittäjätyökaluihin kuuluvasta CPU-profiilin nauhoituksesta, joka näytti kuinka paljon resursseja kukin kirjasto varasi testianimaation pyörittämiseen. Muistakin selaimista löytyi vastaavia toimintoja, mutta niistä saadut mittaustulokset eivät olleet suoraan vertailtavissa Chromen nauhoittamiin CPU-profiileihin. Lopulta testianimaatioista päädyttiin mittaamaan fps-lukujen lisäksi vain Chromen resurssinkäyttö, sillä se tuntui tähän testiin riittävältä mittapuulta. JavaScriptin suoritustehon yksiselitteinen mittaaminen on lopulta hankalaa, sillä mitatut tulokset saattavat vaihdella huomattavasti jo saman selaimen eri versionumeroiden välillä. Tärkeintä oli lopulta selvittää, mikä kirjastoista olisi helppokäyttöisin ja tutkia, aiheutuuko kirjastoista merkittäviä yhteensopivuus- tai suorituskykyongelmia tietyillä alustoilla.

Testianimaatioita varten kirjoitetun koodin ei ollut tarkoitus olla erityisen tehokasta tai kaunistakaan, vaan tarkoituksena oli ennemminkin perehtyä kunkin kirjaston toimintalogiikkaan ja saada nopeasti tuloksia näytölle. Koodinäytteiden ei ole tarkoitus olla esimerkkinä kirjastojen parhaista käytänteistä, vaan pikemminkin havainnollistaa niiden yksilöllisiä eroja samankaltaisen animaation toteutuksessa.

3.3 Testatut grafiikkakirjastot

3.3.1 Processing.js

Processing.js on JavaScript-implementaatio Processing-ohjelmointikielestä, jonka painopisteenä on juuri grafiikan luonti ja interaktiivisuus. Yksinkertaisen rakenteensa ja grafiikkafunktioidensa vuoksi Processingia käytetään jonkin verran ohjelmoinnin opetuksessa, joten se vaikutti aluksi soveltuvan yksinkertaisen ja tiiviin koodin kirjoittamiseen.

Ongelmaksi kuitenkin osoittautui se, että alkuperäinen Processing-kieli on itse asiassa toimintalogiikaltaan melko samanlaista kuin canvasin suora käsittely. Esimerkiksi useamman kappaleen samanaikainen pyöritys vaatii eri objektien transformaatiomatriisien välillä siirtymistä pyörityskomentojen välissä, joten käytännössä koodi sisältää edelleen paljon kontrollitoimintoja, jotka eivät suoranaisesti näy loppukäyttäjälle. Kirjastoa hyödyntävä koodi ei näin ollen ollut merkittävästi tiiviimpää tai selkeämpää kuin canvasia suoraan käsittelevä koodi.

Alla olevaa koodia on selkeyden vuoksi yksinkertaistettu siten, että pyöriviä bittikarttoja on vain yksi. Fps-mittaria ei myöskään ole sisällytetty tähän koodiin. Huomionarvoista on myös se, että koodi ei sisällä lainkaan JavaScriptiä, vaan Processing.js tulkkaa grafiikkaoperaatiot canvasille suoraan Processing-kielestä.

```
/* @pjs preload="img/cage.png,img/jeff.png";*/

int X, Y;
int headRotate;

void setup()
{
  size(480,320);
  X = width/2;
  Y = height/2;
}

PImage head = loadImage("img/cage.png");
Pimage background = loadImage("img/jeff.png");

void draw(){
  headRotate = sin(frameCount/4)*8;
```

```
    pushMatrix();
    imageMode(CENTER);
    image(background,X,Y,480,320);
    popMatrix();

    pushMatrix();
    translate(X+cos(frameCount/24)*48-128,
    Y+sin(frameCount/24)*48-64);

    rotate(radians(headRotate));
    scale(1+sin(frameCount/4)*.05);
    tint(255,192+cos(frameCount/8)*64);
    image(head,0,0,128,128);
    popMatrix();
}
```

Processing.js:ssä bittikarttojen esilataus on hyvin vaivatonta. Alun kommenttilohkoon on sisällytetty ladattavien tiedostojen sijainnit, joista ne saadaan myöhemmin ladattua loadImage()-metodilla.

Draw()-funktio sisältää jokaisella ruudunpäivityskerralla suoritettavat toiminnot, joten koodin suoritusjärjestyksestä tulee hyvin selkeä. Ensin ladataan tarvittavat resurssit, jonka jälkeen siirrytään ajamaan piirto- ja ruudunpäivitysrutiinia. PushMatrix()- ja popMatrix()-metodeilla siirrytään eri transformaatiomatriisien välillä, eli käytännössä ne mahdollistavat eri pyöritys-, skaala- ja sijaintiarvojen antamisen eri graafisille objekteille. Arvot määritellään ennen objektin lisäämistä canvasille, bittikartan tapauksessa image()-metodia käyttäen.

3.3.2 KineticJS

KineticJS on JavaScript-kirjasto, joka laajentaa canvasin tapahtumankäsittelyä ja perustuu päällekkäisten layerien eli kerrosten käyttämiseen. Näiden ominaisuuksiensa vuoksi sitä on käytetty jonkin verran peligrafiikan ohjelmointiin, jonka tarpeet ovat melko samanlaisia kuin opinnäytetyössäkin.

Kerrosten käyttäminen on erottamaton osa KineticJS:ää, sillä canvasin alimmalle tasolle ei ole mahdollista lisätä muun tyyppisiä objekteja kuin kerroksia. Canvasille siis lisätään aluksi kerroksia, joille vuorostaan lisätään niihin piirrettävät objektit. Tämä

tuntui teoriassa rajoitteelta, mutta käytännössä kerrosten käyttäminen oli yksinkertaista eikä vaatinut paljoa ylimääräisen koodin kirjoittamista.

KineticJS ei sisällä omia esilataustoimintojaan, joten sille jouduttiin kirjoittamaan kuvien esilatausta varten oma funktionsa. Tätä ei ole sisällytetty alla olevaan koodinäytteeseen, sillä se ei havainnollista itse kirjaston toimintaa. Selvennettäköön tosin, että ennen drawImages()-funktion ajamista esiladatut kuvat ovat viitattuina images-nimisessä taulukossa. Koodi on myös selkeyden ja tiiviyden nimissä taas typistetty siten, että taustan päällä on vain yksi pyörivä objekti.

```
function drawImages(images) {
  var stage = new Kinetic.Stage({
    container: 'kineticjs',
    width: 480,
    height: 320
  });
  var bg = new Kinetic.Layer();
  var head = new Kinetic.Layer();

  var jeffImg = new Kinetic.Image({
    image: images.jeff,
    width: 480,
    height: 320
  });

  jeffImg.setPosition(0,0);
  bg.add(jeffImg);

  var sprite = new Kinetic.Image({
    image: images.cage,
     offset: [160, 160],
    width: 320,
    height: 320
  });
  sprite.setPosition(stage.getWidth()/2,
  stage.getHeight()/2);
  head.add(sprite);

  stage.add(bg);
  stage.add(head);

  var anim = new Kinetic.Animation(function(frame) {
    sprite.setRotation(frame.time/500);
    sprite.setScale(1.5+Math.sin(frame.time/600)/4);

    sprite.setOpacity(.25+
    Math.abs(Math.cos(frame.time/1000))/4);

  }, head);
```

```
    anim.start();
}
```

Yksi KineticJS:n merkittävä ero Processingiin nähden on oliomaisempi lähestymistapa, jossa näytölle piirrettävät objektit ovat olioita, jotka sisältävät omat funktionsa eri transformaatioita varten. KineticJS:llä työskentely oli siis hyvin samanlaista kuin esimerkiksi grafiikan ohjelmointi ActionScriptillä.

Piirto- ja ruudunpäivitysrutiinien suhteen KineticJS noudattaa eräänlaista mustan laatikon periaatetta, jossa myös animaatioille on oma luokkansa. Animaatio voidaan käynnistää yksinkertaisesti kutsumalla kerran sen start()-metodia, jonka jälkeen se pyörii itsestään kunnes toisin määrätään. Tarvetta omien jokaisella ruudunpäivityksellä ajettavien funktioiden kirjoittamiseen ei siis ollut. Pikaisen kokeilun perusteella järjestelmä on toimiva, ja täyden kontrollin poistaminen ohjelmoijan käsistä voi joissain tapauksissa myös estää huonosti optimoidun koodin kirjoittamista.

3.3.3 EaselJS

EaselJS on osa laajempaa CreateJS-tuotesarjaa, joka kattaa useita JavaScriptin interaktiivisuutta ja multimedia-ominaisuuksia laajentavia kirjastoja. Näistä EaselJS keskittyy grafiikan ja tapahtumankäsittelyn laajennuksiin, ja KineticJS:n tavoin se mahdollistaa oliomaisemman canvas-grafiikan käsittelyn.

CreateJS sisältää myös PreloadJS-nimisen kirjaston resurssien esilatausta varten, joka otettiin käyttöön testianimaatiota varten. Samaan tuotesarjaan kuuluvat EaselJS ja PreloadJS toimivat luonnollisesti hyvin yhdessä, eikä ylimääräisen kirjaston käyttöönotto vaatinut käytännössä enempää dokumentaatioon perehtymistä kuin esilatauksen implementointi aiemmin testaamillani kirjastoilla. PreloadJS:stä tarvittiin yksinkertaista esilatausta varten pelkästään LoadQueue-luokkaa, jonka tapahtumankäsittely mahdollisti funktioiden ajamisen resurssien latausten valmistuttua.

```
var queue = new createjs.LoadQueue();
queue.on("complete", handleComplete, this);
queue.loadManifest([
     {id: "bgImg", src:"img/jeff.png"},
     {id: "headImg", src:"img/cage.png"}
```

```
]);

var bgImg;
var headImg;
```

Itse EaselJS muistuttaa vielä KineticJS:ääkin läheisemmin grafiikkaohjelmointia ActionScriptillä - kun KineticJS:ssä objektien transformaatiot toteutettiin kutsumalla niiden set-metodeja (esim. object.setRotation(90)), EaselJS:n objektien muuttujille voidaan suoraan antaa arvoja (object.rotation = 90) jotka päivittyvät asianmukaisesti näytölle. Kuten aiemmissakin esimerkeissä, tässä koodissa piirretään näytölle vain yksi pyörivä objekti.

```
function handleComplete() {
  bgImg = queue.getResult("bgImg");
  headImg = queue.getResult("headImg");

  stage = new createjs.Stage("demoCanvas");

  var bg = new createjs.Bitmap(bgImg);
  bg.x = 0;
  bg.y = 0;
  bg.scaleX = 2;
  bg.scaleY = 2;
  stage.addChild(bg);

  var sprite = new createjs.Bitmap(headImg);
  sprite.scaleX = 8;
  sprite.scaleY = 8;
  sprite.regX = headImg.width / 2;
  sprite.regY = headImg.height / 2;
  sprite.x = stage.canvas.width / 2;
  sprite.y = stage.canvas.height / 2;
  stage.addChild(sprite);

  createjs.Ticker.addEventListener("tick", handleTick);
  createjs.Ticker.setFPS(60);

  var rotSpeed = 4;

  function handleTick(event) {
    sprite.alpha =
    .5+Math.sin(createjs.Ticker.getTicks() / 20) / 8;

     sprite.scaleX +=
    Math.sin(createjs.Ticker.getTicks() / 30) / 60;

     sprite.scaleY +=
    Math.sin(createjs.Ticker.getTicks() / 30) / 60;

     sprite.rotation += rotSpeed / 2;
```

```
            stage.update();
        }
    }
```

EaselJS:n ruudunpäivitys vaatii hieman enemmän omatoimisuutta kuin KineticJS:n yhdellä komennolla käynnistyvät animaatiot. Kirjaston oma Ticker-olio laskee ajan kulumista sovelluksessa, ja siihen voidaan liittää tapahtumankäsittelijä animaatioiden ja ruudunpäivityksen ohjelmoimista varten.

Kaikista testatuista kirjastoista EaselJS tuntui helppokäyttöisimmältä, osin siksi että se muistutti rakenteeltaan eniten ActionScriptiä. EaselJS tuntui myös tarjoavan parhaan tasapainon valmiiksi ohjelmoitujen mustien laatikoiden sekä vapaamman rutiinien kirjoituksen välillä.

3.4 Grafiikkakirjastojen suoritustehojen vertailu

Testimenetelmiä kuvattaessa pantiin merkille, että JavaScriptin suoritustehon mittaamisessa on omat haasteensa. Eri laitealustat ja selainversiot voivat vaikuttaa hyvinkin paljon tuloksiin, mittausmenetelmät eivät ole absoluuttisen tarkkoja ja paperilla vastaavanlaisilta vaikuttavat tulokset voivat olla näytöllä silminnähden erilaisia.

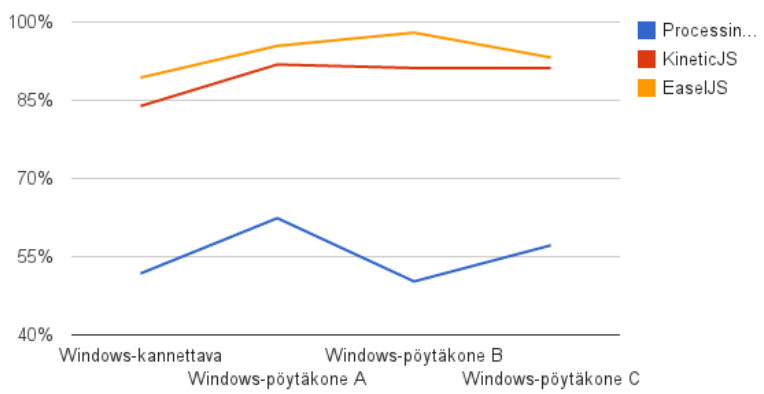
Edes osin vertailukelpoisten tuloksien saamiseksi testianimaatioiden fps-luvut mitattiin eri laitteilla ja selaimilla. Chromella tehdyissä testeissä nauhoitettiin myös koneen CPU-profiili animaation ajalta, jotta kunkin kirjaston vaatimista resursseista saataisiin fps-lukua tarkempaa näyttöä.

Taulukossa 1 (sivu 28) näkyvät eri laitteilla ja selaimilla mitatut fps-luvut eri grafiikkakirjastoilla toteutetuista testianimaatioista. Taulukossa olevat luvut ovat karkeita keskiarvoja, eikä niistä ilmene lukujen mahdollista vaihtelua animaation aikana. KineticJS ja EaselJS saavuttivat miltei jokaisella testikerralla täyden 60 fps:n ruudunpäivityksen, kun taas Processing.js oli altis matalammille lukemille. Erityisen huomionarvoisia ovat iPad 2:lla mitatut lukemat, joissa Processing.js:n ruudunpäivitys tipahti yksilukuisiin arvoihin muiden grafiikkakirjastojen toimiessa täydellä vauhdilla.

Taulukko 1. Testianimaatioiden fps-luvut eri laitteilla ja selaimilla.

Laite	Selain	Processing.js	KineticJS	EaselJS
iPad 2	Safari	7	60	60
Windows-kannettava	Chrome v.36	60	60	60
Windows-kannettava	Firefox v.22	60	60	60
Windows-kannettava	Internet Explorer 11	60	60	60
Windows-pöytäkone A	Chrome v.33	60	60	60
Windows-pöytäkone A	Firefox v.28	60	60	60
Windows-pöytäkone A	Internet Explorer 11	60	60	60
Windows-pöytäkone B	Chrome v.33	59	60	60
Windows-pöytäkone B	Firefox v.27	37	60	60
Windows-pöytäkone B	Internet Explorer 10	42	60	60
Windows-pöytäkone C	Chrome v.34	60	60	59
Windows-pöytäkone C	Firefox v.28	55	60	60
Windows-pöytäkone C	Internet Explorer 11	60	60	60

Kuviossa 4 ovat Chromella mitattujen CPU-profiilien tulokset samoilta laitteilta, lukuun ottamatta iPadia, jolla tällaista ominaisuutta ei ollut saatavilla. Prosenttiluvut viittaavat siihen laskentaresurssien määrään, joka jäi animaation aikana vapaaksi – suurempi luku viittaa siis pienempään resurssien varaukseen ja parempaan suorituskykyyn.



Kuvio 4. Testianimaatiossa vapaaksi jääneet CPU-resurssit eri laitteilla.

Kuten testituloksista näkyy, Processing.js:n suoritusteho oli selkeästi huonoin testatuista kirjastoista. Processing-kielen ohjelmointikäytänteet eivät myöskään käytännössä juuri helpottaneet canvas-animaatioiden ohjelmointia, joten se jouduttiin toteamaan opinnäytetyön tarkoituksiin soveltumattomaksi. Jälkikäteen vaikuttaisikin siltä, että Processing.js soveltuisi parhaiten valmiiden Processing-sovellusten web-versioiden toteuttamiseen.

KineticJS oli suoritusteholtaan jo selkeästi parempi. Paperille kirjatuista tuloksista ei kuitenkaan ilmene, että se oli yllättävän altis ruudunpäivityksen äkillisille laskuille ja piikeille. Tämä johti animaatiota tehokkaastikin pyörittäneissä koneissa epätasaiseen ja nykivään animaatioon. Yhdessä testeistä Internet Explorer 11:ssa ilmeni myös yhteensopivuusongelmia testianimaation kanssa: ruudunpäivitys toimi vain yhdessä osassa canvasia, jonka ulkopuolella kuva ei päivittynyt ja liikkuvat objektit jättivät jälkeensä haamukuvia itsestään. Nämä ongelmat saattoivat periaatteessa johtua myös testianimaatioon jääneistä ohjelmointivirheistä, mutta lopulta opinnäytetyössä päädyttiin käyttämään mieluummin EaselJS:ää kuin käyttämään ylimääräistä aikaa KineticJS:n yhteydessä ilmenneiden hankaluuksien korjaamiseen.

Lopulta EaselJS oli opinnäytetyöhön sopivin ja myös testeissä parhaan suoritustehon saavuttanut grafiikkakirjasto. Muihin kirjastoihin käytetty aika oli kuitenkin hyödyksi koko opinnäytetyön kannalta, sillä testianimaatioiden toteuttaminen eri menetelmillä antoi paljon pohdittavaa hyvistä yleisistä periaatteista grafiikan ja animaatioiden ohjelmointiin.

4 NestBotin tuotanto

4.1 ELIZA esikuvana ja lähdemateriaalina

Historiallisista tekoälyistä erityisen merkittäväksi ja kiinnostavaksi luvussa 2.4.3 kuvailtu ELIZA. Se ei ollut toiminnaltaan erityisen sofistikoitunut tekoäly, mutta se sai silti käyttäjänsä uskomaan niin. Opinnäytetyön tekoälyn suoraksi esikuvaksi valittiin siis ELIZA, jonka toimintaan oli näin ollen perehdyttävä tarkemmin.

Norbert Landsteinerin (2005) Elizabot.js on JavaScriptillä toteutettu versio alkuperäisestä ELIZAsta. Sen lähdekoodista oli mahdollista selvittää, kuinka ELIZA käsittelee käyttäjältä saatua syötettä ja kuinka se muodostaa sen pohjalta oman tulosteensa. Se tarjosi myös mahdollisuuden testata välittömästi, miten ELIZA käytännössä vastaa tietynlaisiin syötteisiin.

4.2 ELIZAn lauseenkäsittely

Lyhyesti kuvailtuna ELIZA vertaa käyttäjältä saatua syötettä sen muistissa olevaan listaan erilaisista avainfraaseista. Eri fraaseilla on omat prioriteettiarvonsa siltä varalta, että syöte sisältää useamman kuin yhden avainfraasin. ELIZAlla on jokaista fraasia kohden luettelo vastauksista, joista se valitsee umpimähkään yhden tulostettavaksi. Jos käyttäjä siis tervehtii ELIZAa kirjoittamalla “hello”, ELIZA vastaa käyttäjälle jollakin niistä tulosteista, jotka sen muistissa on liitetty avainfraasiin “hello”.

Käytännössä tämä vaatii jonkin verran syötteen käsittelyä erilaisten erityistilanteiden varalta. Pelkkä yksinkertainen merkkijonovertailu saattaisi johtaa esimerkiksi tilanteeseen, jossa ELIZA tulkitsisi syötteen “my favorite band is Helloween” tervehdykseksi, koska se sisältää “hello”-merkkijonon. Käyttäjä saattaa myös käyttää jotakin aivan toisenlaista tervehdystä, kuten “hi” tai “greetings”. Kirjainkootkin tulee ottaa huomioon, sillä koneellisesti vertailtuna “HELLO” ja “hello” ovat jo täysin erilaisia merkkijonoja.

Tällaisten tapausten varalta ELIZA muuntaa saamansa syötteen kokonaan pieneen kirjainkokoon, poistaa siitä erikoismerkkejä ja muuntaa erilaiset välimerkit pisteiksi yksinkertaisempaa lauseenkäsittelyä varten. Syötettä verrataan myös ELIZAn muistissa olevaan synonyymiluetteloon, johon sisällytetyt sanat voidaan korvata ELIZAn ymmärtämillä avainsanoilla. Esimerkiksi sanat “unhappy”, “depressed” ja “sick” korvataan lauseenkäsittelyvaiheessa aina sanalla “sad”, joka löytyy ELIZAn avainfraasiluettelosta.

ELIZA pystyy myös käyttämään käyttäjän antamaa syötettä osana omaa tulostettaan. Esimerkiksi syötteeseen “I feel like a new man” ELIZA saattaa vastata “Do you enjoy feeling like a new man?”. Jotkin ELIZAn vastauksista sisältävät tätä varten kontrollikoodeja, jotka korvataan ennen vastauksen antamista avainfraasin jälkeen tulevalla syötteellä. Yllä annetussa tapauksessa ELIZAn avainfraasien joukosta löytyy “I feel”, jonka vastaukseksi valitun tulosteen muoto on koodissa vielä “Do you enjoy feeling (2)?”. Vastauksen kohta “(2)” korvataan avainfraasin jälkeisellä osalla käyttäjän antamasta merkkijonosta, tässä tapauksessa siis “like a new man”.

Käyttäjän antaman syötteen heijastamisesta syntyy kuitenkin uusi ongelma. Syöte saattaa sisältää persoonapronomineja, jotka eivät sellaisenaan sovi järkevästi ELIZAn tuottamiin lauseisiin. Esimerkiksi syöte “I feel like I need to work harder” tuottaisi pelkästään yllä kuvatulla algoritmilla vastauksen “Do you enjoy feeling like I need to work harder?” Tätä varten ELIZAlla on vielä luettelo sanoista, jotka korvataan käyttäjän antamasta syötteestä ennen sen liittämistä osaksi ELIZAn omaa vastausta. Luettelo koostuukin enimmäkseen ensimmäisen persoonan pronomineista sekä niiden toisen persoonan vastineista.

Jos käyttäjän syöte ei sisällä ainuttakaan ELIZAn tunnistamaa avainsanaa, ohjelma syöttää itselleen avainfraasiksi tätä tilannetta vastaavan kontrollikoodin. Mielenkiintoista on, että ELIZAn vastaukset tällaisiin tilanteisiin ovat enimmäkseen kehotuksia kertomaan asiasta lisää, ikään kuin tekoäly ymmärtäisi käyttäjän syötteen täysin.

Tällaisilla menetelmillä ELIZA saa aikaiseksi melko loogisia vastauksia. Loppujen lopuksi sillä on kuitenkin omat rajoitteensa, ja hetken määrätietoisella kokeilulla on

mahdollista löytää syötteitä, joihin se antaa kieliopillisesti sekavia tai ainakin terapeutille sopimattomia vastauksia. Esimerkiksi syötteeseen “Someone told me that I am ugly and fat” ELIZA saattaa vastata “Do you enjoy being ugly and fat?”

Alla on esimerkki yhdestä ELIZAn kanssa käydystä keskustelusta, joka havainnollistaa joitakin sen yllä kuvatuista toimintatavoista. Keskusteluun on merkitty lihavoituna ne avainsanat, joiden pohjalta ELIZA on valinnut antamansa vastauksen. Keskustelusta voidaan myös panna merkille, kuinka Eliza toistelee omissa vastauksissaan osia käyttäjän antamasta syötteestä, paikoin hieman mekaanisesti ja huonolla kieliopilla.

ELIZA: How do you do. Please tell me your problem.
*YOU: **I am** under a lot of stress.*
ELIZA: Do you know anyone else who is under a lot of stress ?
*YOU: **Yes**, several of my classmates.*
ELIZA: I understand.
*YOU: Many of us **are** working on our theses now, you see.*
ELIZA: Did you think they might not be working on our theses now ?
*YOU: **Your** grammar is incorrect.*
ELIZA: What makes you think of my grammar is incorrect ?

4.3 NestBotin lauseenkäsittely

4.3.1 Uusi sovellus ELIZAn pohjalta

Yllä kuvattiin karkeasti se prosessi, jonka ELIZA käy lävitse käsitellessään käyttäjän syötettä. ELIZA oli opinnäytetyön tekoälyn pääasiallinen esikuva, joten siihen pyrittiin ohjelmoimaan mahdollisimman samankaltainen lauseenkäsittelyalgoritmi. Ensimmäiset yritykset olivat yksinkertaisia prototyyppejä, jotka vain tunnistivat avainsanoja käyttäjän antaman syötteen seasta tai korvasivat siinä esiintyneitä avainsanoja toisilla sanoilla. Ajan myötä yksittäiset, konseptien testaamiseksi kirjoitetut lauseenkäsittelyfunktiot alkoivat muodostua yhtenäiseksi kokonaisuudeksi.

Ohjelmaa kirjoittaessa törmättiin moniin yllä kuvattuihin ongelmatilanteisiin, ja ELIZAn lähdekoodia jouduttiin usein tutkimaan uudelleen ja sen toimintaa testattiin

useissa eri tilanteissa. Tarkoituksena ei kuitenkaan ollut ottaa ELIZAn lähdekoodista suoria lainauksia, vaan ennemminkin kokeilla itse vastaavanlaisen järjestelmän tekemistä. NestBotin lauseentulkinta onkin lopulta hieman yksinkertaisempaa kuin ELIZAn, eikä se suoranaisesti tee mitään mihin ELIZA ei pystyisi. Näin pysyttiin kuitenkin hyvin perillä siitä, mitä ohjelman jokaisella koodirivillä tapahtuu ja animaatioiden sekä muiden ELIZAsta puuttuvien ominaisuuksien lisääminen oli helpompaa.

4.3.2 Sanasto - avainsanat, synonyymit ja korvattavat sanat

NestBotin pohjatietoina toimii joukko lujakoodattuja sanakirjoja, joita käytetään syötteiden tulkintaan ja käsittelyyn. NestBotin sanakirjojen rakenne ja toimintalogiikka muuttuivat jonkin verran tekoälyn suunnittelun ja ohjelmoinnin aikana, ja lopulta ne jaettiin kolmeen taulukkomuotoiseen muuttujaan: avainsanoihin, synonyymeihin ja korvattaviin sanoihin.

NestBotin tunnistamat avainsanat on tallennettu kaksiulotteiseen taulukkoon nimeltä “responses”, jossa avaimena on aina itse avainsana merkkijonona ja solun arvona yksi tai useampi mahdollinen vastaus kyseiseen avainsanaan. Kun taulukon avaimena käytetään haettavaa merkkijonoa, käyttäjän syötettä voidaan vertailla suoraan sanastossa oleviin avainsanoihin. Taulukossa 2 on esimerkki responses-taulukon rakenteesta:

Taulukko 2. Esimerkki avainsanojen ja vastauksien taulukoinnista.

Avainsana	**Vastaukset**		
“hello”	“Greetings!”		
“help”	“Don’t worry.”	“I am here to help you.”	
“thanks”	“You’re welcome.”	“No problem!”	“No sweat!”

Vastaukset voivat sisältää myös kontrollikoodeja, jotka mahdollistavat käyttäjän syötteen sisällyttämisen vastaukseen tai funktioiden kutsumisen vastauksen yhteydessä.

Kontrollikoodit ovat osa samaa merkkijonoa kuin itse vastauskin, mutta ne poistetaan käyttäjälle näkyvästä tulosteesta.

Koodeista ensimmäinen on “_post_”, joka korvataan ennen vastauksen tulostamista sillä syötteen osalla, joka seuraa avainsanaa. Jos siis avainsanan “I am” vastaus olisi “I heard that you are _post_”, syöte “I am very handsome” saisi NestBotin vastaamaan “I heard that you are very handsome”.

Toinen kontrollikoodi on “_function_”, joka kutsuu sitä funktiota, joka nimetään kontrollikoodin jälkeen. Funktion nimen perään voidaan myös lisätä “_arg_”-kontrollikoodi, jonka perään voidaan puolestaan kirjoittaa funktiolle välitettävä parametri. Näin ollen esimerkiksi vastaus “Error!_function_setScreenColor_arg_red” antaisi käyttäjälle vastauksen “Error!” ja kutsuisi samalla funktion setScreenColor(red).

NestBot sisältää myös listan synonyymeistä, eli ohjelmalogiikan kannalta katsottuna sellaisista sanoista, jotka vastaavat jotakin NestBotin tunnistamaa avainsanaa. Myös synonyymit ovat tallennettu kaksiulotteiseen taulukkoon, jonka nimi on “synonyms”. Taulukon avaimena toimii jokin responses-taulukon avainsanoista, ja solujen arvoina on luettelo niistä sanoista jotka vastaavat tätä avainsanaa. Taulukossa 3 on esimerkki synonyms-taulukon rakenteesta:

Taulukko 3. Esimerkki synonyymien taulukoinnista.

Avainsana	**Synonyymit**		
“i am”	“i’m”		
“friend”	“pal”	“bro”	
“hello”	“hi”	“hey”	“greetings”

Kolmas ja viimeinen sanasto sisältää sellaiset sanat, jotka korvataan NestBotin antamissa vastauksissa joillakin muilla sanoilla. Kuten ELIZAssakin, tällä pyritään vaihta-

maan käyttäjäsyötteessä olevia pronomineja siten, että vastaukset pysyvät johdonmukaisina. Korvattavat sanat on tallennettu yksiulotteiseen, “replace”-nimiseen taulukkoon, jonka avaimina toimivat syötteestä korvattavat sanat ja arvoina avaimen paikalle tulevat uudet sanat. Taulukossa 4 on esimerkki replace-taulukon rakenteesta:

Taulukko 4. Esimerkki korvaussanojen taulukoinnista.

Korvattava sana	**Korvaava sana**
“you are”	“I am”
“me”	“you”
“my”	“your”

Jokainen yllä kuvatuista taulukoista on käytännössä järjestelty tärkeysjärjestykseen siten, että aikaisemmin taulukossa esiintyvillä avainsanoilla on korkeampi prioriteetti. Tämä on hieman karumpi ratkaisu kuin ELIZAn sanastossa, jossa avainsanoille oli erikseen määritettynä oma numeromuotoinen prioriteettinsa, mutta toisaalta NestBotin sanasto ja sen priorisointi on näin selkeämmin ymmärrettävissä yhdellä vilkauksella.

Yleisesti ottaen tärkeämmiksi on määritetty sellaiset avainsanat, jotka esiintyvät käyttäjän syötteessä harvemmin ja viittaavat johonkin merkitykselliseen asiaan. Alinta prioriteettia edustavat yleiset kieliopilliset rakenteet. Esimerkiksi avainsanaluettelossa “artificial intelligence” on listan yläpäässä, kun taas “I” on sen alapäässä. Jos käyttäjä siis antaa syötteekseen “I am interested in artificial intelligence”, NestBot reagoi avainsanaan “artificial intelligence” ja antaa siihen liittyvän vastauksen sen sijasta, että se kiinnostuisi syötteen alussa olevasta “I”-sanasta.

4.3.3 Lauseen esikäsittely

NestBot muokkaa heti ensimmäiseksi käyttäjän antamaa syötettä koneellisen tulkinnan helpottamiseksi. Syöte muutetaan aluksi kokonaan pieneen kirjainkokoon, jonka jälkeen kaikki sen sisältämät välimerkit korvataan pisteillä.

Tämän jälkeen tekstistä karsitaan pois potentiaalisesti suurikin osa: kaikki teksti, joka seuraa joko pistettä, “or”-sanaa tai “but”-sanaa poistetaan syötteestä. Toisin sanottuna NestBot ei edes yritä tulkita syötettä sen ensimmäistä päälausetta pidemmälle. Tämä voi vaikuttaa ensi alkuun tarpeettoman suurelta leikkaukselta, mutta käytännössä se johti testitilanteissa tiiviimpien ja järkevämpien vastausten antamiseen. Myös erikoismerkit poistetaan syötteestä, jotta siihen ei voi sisällyttää ohjelmakoodia tai NestBotin omia kontrollikoodeja.

Lopuksi syötteestä etsitään “synonyms”-taulukon arvoja vastaavia sanoja, jotka korvataan NestBotin ymmärtämillä avainsanoilla.

```
function handleString(input) {
  if(input) {
    input = input.toLowerCase();
    input = input.replace(/[!,;:]/g, '.');
    input =
    input.replace(/(\sbut\s|\sor\s|\.)(.|)+/g,'');
    input = input.replace(/[^a-zA-Z0-9\s']/g,'');
    input = input.replace(/ +(?= )/g,'');
    input = " " + input + " ";

    for (var key in synonyms) {
      for (var i=0, len=synonyms[key].length;i<len;i++) {
        input = input.replace(" "+synonyms[key][i]+" ",
        " "+key+" ");

      }
    }
```

NestBotin lauseenkäsittelyä voidaan tässä vaiheessa jo kritisoida siitä, että se ei ole laskennallisesti erityisen tehokas. Syötettä käsitellään usealla peräkkäisellä säännöllisellä lausekkeella, ja kaksiulotteinen synonyymitaulukko käydään jokaisella käsittelykerralla kokonaisuudessaan lävitse. Tarkoituksena oli kuitenkin kirjoittaa mahdollisimman selkeää ja yksinkertaista koodia, jota voidaan tarpeen mukaan kehittää jat-

kossa kiinteämmäksi osaksi FreeNestiä. Käsiteltävät syötteet ovat pääasiallisesti vain muutaman sanan mittaisia, eivätkä ohjelmalle käsin syötetyt sanakirjat tule kovin todennäköisesti paisumaan niin suuriksi, että niiden läpikäynti muodostuisi laskennalliseksi pullonkaulaksi. Lauseenkäsittelyalgoritmin tehokkuus olisi olennaisempaa, jos NestBot olisi oppiva tekoäly, joka esimerkiksi rakentaisi Markovin ketjuja käyttäjän antamasta syötteestä. Tällaisen ohjelman omatoimisesti rakentaman sanakirjan koko saattaisi hyvinkin kasvaa ajan myötä todella suureksi.

4.3.4 Avainsanojen tunnistaminen

Luvussa 4.3.2 mainittiin, että NestBotin sanastojen sisältämät avainsanat ovat järjestetty prioriteetin mukaan. Avainsanojen tunnistaminen vaatii näin ollen vain avainsanataulukon läpikäymisen ja syötteestä löytyneiden avainsanojen tallentamiseen omaan taulukkoonsa.

```
var matches = new Array();
  for (var key in responses) {
    if(input.match(" "+key+" ")) {
      matches.push(input.match(key));
    }
  }
if(matches.length==0) matches[0] = "_none_";

var match;
if(randomResponse == true) {
  match =
  matches[Math.floor(Math.random()*matches.length)]

} else {
  match = matches[0];
}
```

Jos syötteestä ei löydy ainuttakaan avainsanaa, taulukon ensimmäisen solun sisällöksi pakotetaan tähän tilanteeseen varattu avainsana “_none_”. Avainsanaan ei liity mitään erityisiä funktioita, vaan sen yhteyteen on vain liitetty omat vastauksensa.

NestBot voi valita kahdella eri tavalla sen avainsanan, jonka perusteella se valitsee vastauksensa. Jos randomResponse-muuttuja on tosi, NestBot valitsee umpimähkäi-

sesti jonkin lauseesta löytyneistä avainsanoista. Muussa tapauksessa valitaan se avainsana, jonka prioriteetti on korkein.

4.3.5 Vastauksen muodostaminen

Kun NestBot on valinnut käsiteltävän avainsanan, se ryhtyy muodostamaan sen perusteella vastaustaan käyttäjälle. Itse vastauksen valinta vaatii vain satunnaisen solun valinnan “responses”-taulukosta valitun avainsanan alta. Vastaukselle täytyy kuitenkin vielä ajaa joitakin toimintoja, ennen kuin siitä saadaan tilanteeseen sopiva ja järjellinen.

```
var response =
responses[match][Math.floor(Math.random()
* responses[match].length)];

var post = input.split(" "+match+" ").pop().trim();

for (var key in replace) {
  var regex = new RegExp("\\b"+key+"\\b","g");

  post = post.replace(regex,
  "_replaced_"+replace[key]+"_replaced_");
}

while (post.match("_replaced_")) {
  post = post.replace("_replaced_","");
}
response = response.replace("_post_",post);

response = response.charAt(0).toUpperCase()
+ response.slice(1);

response = response.replace(/\sI\./g,' me.');
```

Aiemmin kuvailtiin erilaisia kontrollikoodeja, joita sanastoon tallennettuihin vastauksiin voidaan sisällyttää. Käyttäjän antamasta syötteestä tallennetaan aina valitun avainsanan jälkeinen osa omaan “post”-nimiseen muuttujaansa, jonka sisällöllä korvataan juuri joidenkin vastausten sisältämä “_post_”-kontrollikoodi.

Aiemmin kerrottiin myös, kuinka “replace”-taulukon avaimista löytyvät sanat korvataan soluarvojen sanoilla pääasiassa pronominien persoonan vaihtamiseksi. Tässä käytetään säännöllistä lauseketta, joka suuntaa nämä muutokset vain kokonaisiin

sanoihin. Näin ollen jos käyttäjän syöte sisältää esimerkiksi sanan “isinglass”, sitä ei muuteta muotoon “yousyouglass”. Korvaukset suoritetaan myös ainoastaan “post”-muuttujaan tallennetulle merkkijonolle, jotta vastausten lujakoodattu osa ei muutu lauseenkäsittelyn aikana.

Korvattujen sanojen alkuun ja loppuun lisätään myös väliaikaisesti “_replaced_”-merkkijono. Ohjelman käynnistysvaiheessa “replace”-taulukon arvoista korvataan kaikki välilyönnit samalla merkkijonolla. Tällä estetään se, ettei ohjelma ala korvaamaan jo kerran korjattuja sanoja rekursiivisesti. Jos tällaista toimenpidettä ei tehtäisi, ohjelma saattaisi esimerkiksi ensin korvata sanat “I am” sanoilla “you are”, mutta palauttaa ne entiselleen korvatessaan vuorostaan sanoja “you are” sanoilla “I am”. Kun kaikki “replace”-taulukossa määritetyt korvaukset on tehty, vastauksesta poistetaan kaikki “_replaced_”-merkkijonot.

Vastaukselle suoritetaan vielä kaksi lyhyttä varmuustoimenpidettä ennen sen tulostamista. Vastaukselle pakotetaan iso alkukirjain siltä varalta, että sillä ei sellaista jo ole. Lisäksi jos vastaus päättyy “I”-sanaan, se korvataan “me”-sanalla. Lauseen lopussa olevan ensimmäisen persoonan pronominin olisi todennäköisesti tarkoitus olla objektimuodossa, joten korvaus tehdään tällaisten tilanteiden varalta. Ratkaisu on ajatustasolla melko karu, mutta se estää yleensä “are you talking to I” -tyyppisten vastausten antamisen eikä häiritse ohjelman muuta toimintaa.

NestBotin vastauksiin voidaan myös sisällyttää niiden aikana ajettavia funktioita luvussa 4.3.2 kuvaillulla tavalla. Ajettavat funktiot siis nimetään osana merkkijonoa, mikä vaatii muun muassa potentiaalisten tietoturvariskien vuoksi hieman monimutkaisempaa implementaatiota.

```
if (response.match("_function_")) {
  var functionName =
  response.slice(response.indexOf("_function_"))
  .replace("_function_","");

  var functionArg =
  functionName.slice(functionName.indexOf("_arg_"))
  .replace("_arg_","");

  functionName =
  functionName.slice(0,functionName.indexOf("_arg_"));
```

```
    response =
    response.replace("_function_"+functionName,"")
    .replace("_arg_"+functionArg,"");

    var fn = window[functionName];
    if (typeof fn === "function") fn(functionArg);
  }
```

Kun vastauksesta on eristetty funktion nimeä ja argumentteja vastaavat merkkijonot omiksi muuttujikseen, siitä poistetaan funktioon viittaavat kontrollikoodit. Seuraavaksi tarkistetaan vielä, löytyykö koodista tai JavaScriptin metodeista nimettyä funktiota, jonka jälkeen funktio voidaan ajaa. Aiemmin mainittiin, että käyttäjän antamasta syötteestä poistetaan kaikki erikoismerkit syötteen esikäsittelyvaiheessa. Käyttäjä ei siis pysty omalla syötteellään määräämään ajettavia funktioita, vaan ainoastaan vastauksiin koodattujen funktioiden ajaminen on mahdollista.

4.4 NestBotin grafiikka

4.4.1 Virtuaalihahmon ruumiinrakenne

NestBot-projektissa kyse oli nimenomaan virtuaalihahmon tuottamisesta, joten tekoälylle tarvittiin jonkinlainen huomiota kiinnittävä ja elävästi animoitu graafinen edustus. Kuten aiemmin kerrottiin, sovellukseen valittiin valmiiksi grafiikkakirjastoksi EaselJS sen nopeuden ja helppokäyttöisyyden vuoksi.

Ensimmäisenä tavoitteena oli niin sanotun luurangon luominen animaatiohahmolle. Tällä tarkoitetaan käytännössä järjestelmää, jossa hahmografiikka koostuu useasta toisiinsa liitetystä osasta. Yhden osan liikkuessa myös siihen kiinnitetyt osat liikkuvat sen mukana. Esimerkiksi hahmon olkavarren liikkuessa myös kyynärvarsi lähtee liikkeelle ja pysyttelee asianmukaisesti kiinni olkavarren päässä.

Olennaisinta tällaisen animaatiojärjestelmän toteutuksen kannalta oli EaselJS:n Container-luokan käyttö. Containerit ovat näyttöobjekteja, joiden lapsiksi voidaan määritellä muita näyttöobjekteja, mukaan lukien uusia Containereita. Näin ollen Container-objektin liikkuessa kaikki sen lapsiksi määritellyt näyttöobjektit liikkuvat

sen mukana, mutta niitä voidaan myös liikutella itsenäisesti vaikuttamatta niiden vanhempana toimivaan Containeriin.

Virtuaalihahmon ruumis jaettiin siis seitsemään osaan: päähän, keskivartaloon, jalkoihin, sekä kummankin puoleisiin olka- ja kyynärvarsiin. Ruumiinosat ovat näkyvissä irrallisina toisistaan kuviossa 5.

Kuvio 5. Testihahmon ruumiinosat.

Kunkin ruumiinosan grafiikka tallennettiin omaan kuvatiedostoonsa, jonka jälkeen voitiin ruveta rakentamaan sisäkkäisistä Container-objekteista ja niihin asetelluista bittikartoista koostuvaa tietorakennetta niitä varten. Kullekin ruumiinosalle annettiin oman akselinsa, joka vastasi bittikartoissa sitä pikselikoordinaattia, jonka ympärillä ruumiinosa pyörii ja josta se kiinnittyy muihin ruumiinosiin. Ruumiinosille voidaan antaa myös omat x- ja y-koordinaattinsa, joilla ne sijoitellaan oikeisiin paikkoihinsa. Alla olevaa esimerkkiä on typistetty siten, että hahmo koostuu vain keskivartalosta ja jaloista. Bittikartat on jo ladattu aiemmin kuvatulla EaselJS:n esilatausfunktiolla, ja niihin viitataan “img_”-alkuisissa muuttujissa:

```
var torso = new createjs.Bitmap(img_torso);
torso.regX = img_torso.width / 2;
torso.regY = img_torso.height / 2;
var legs = new createjs.Bitmap(img_legs);
legs.regX = 40;
legs.regY = 6;

var character = new createjs.Container();
character.addChild(torso);

var boneLegs = new createjs.Container();
```

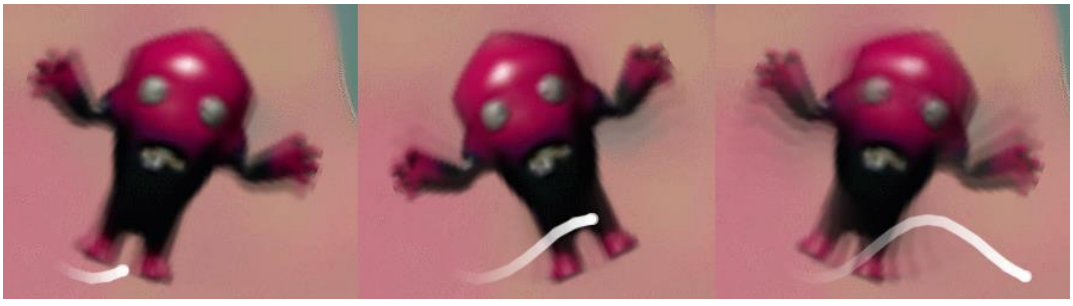
```
boneLegs.addChild(legs);
boneLegs.y = 40;
character.addChild(boneLegs);
```

Näin saatiin aikaiseksi hahmo, jonka ruumiinosien asentoa pystyttiin vaihtamaan koodista käsin. Seuraavaksi hahmo tarvitsi jonkinlaisen animaatiorutiinin, jolla se saataisiin toistamaan jotakin liikerataa ohjelman ajon aikana.

4.4.2 Edestakainen animaatio sini- ja kosinilaskuilla

Grafiikkakirjastojen testianimaatioita kirjoitettaessa pyörivien ja edestakaisin liikkuvien objektien animaatioon käytettiin sini- ja kosinilaskuja. Sinin ja kosinin kuvaajista voidaan nähdä, että tasaisesti etenevillä arvoilla ne muodostavat pehmeää, edestakaista aaltoilua. Animaatiossa on sattumoisin aina yksi tasaisesti kasvava arvo: näytettyjen kuvien eli ruudunpäivitysten lukumäärä.

Jos näytöllä olevalle objektille siis annetaan jonkin transformaation arvoksi senhetkisen ruudunpäivitysten lukumäärän sini tai kosini, kyseinen arvo alkaa aaltoilemaan edestakaisin. Esimerkiksi pyörityksen tapauksessa objekti keinuu puolelta toiselle, kun taas skaalauksessa objekti vuoroin paisuu suuremmaksi ja kutistuu takaisin pienemmäksi. Kuviossa 6 havainnollistetaan, miten objekti keinuu kun sen pyöritysarvoksi annetaan ruudunpäivitysten lukumäärän sini. Valkoisen vanan pää kuvaa siniaallon tilaa kussakin animaation vaiheessa.



Kuvio 6. Siniaallon vaiheen vaikutus objektin pyöritykseen.

Aaltoilun nopeutta ja voimakkuutta voidaan säädellä kertomalla tai jakamalla sinilaskulle annettavaa ruudunpäivitysten määrää ja tämän sinilaskun tulosta. Esimerkiksi kuviossa 6 objektia pyöritellään seuraavanlaisella komennolla:

```
object.rotation = Math.sin(createjs.Ticker.getTicks() /
30) * 25;
```

Sulkujen sisäpuolella oleva numero vaikuttaa siis aallon taajuuteen ja ulkopuolella oleva puolestaan sen amplitudiin.

Haittapuoli animaatioiden tekemisessä pelkillä sineillä ja kosineilla on se, että animaatiot rajoittuvat käytännössä osien edestakaiseen heilutteluun. Menetelmä ei myöskään sovellu hyvin monivaiheisten animaation tekemiseen tai sulaviin siirtymiin erilaisten animaatioiden välillä. Sen etuna on kuitenkin tiiviys ja yksinkertaisuus, sillä jokainen edestakainen liike vaatii vain yhden koodirivin, jota voidaan tarpeen vaatiessa myös muokata nopeasti.

4.4.3 Interaktiivisuus ja animaatioiden välillä siirtyminen

Kun hahmografiikalle ja animaatioille oli omat valmiit järjestelmänsä, seuraava vaihe oli saada hahmo reagoimaan jonkinlaiseen käyttäjäsyötteeseen tietyllä animaatiolla.

Canvasille lisättiin tätä varten nappi, jota painamalla käyttäjä voi “tervehtiä” hahmoa. Nappia painamalla hahmo heiluttaa kättään ja sen pään vierelle ilmestyy puhekupla. Kuviossa 7 on kuvankaappaus hahmosta tervehtimässä käyttäjää valmiissa prototyypissä.

Kuvio 7. Animaatioprototyyppi toiminnassa.

Prototyyppiin tarvittiin tätä varten kaksi animaatiota: paikallaan seisominen ja käyttäjälle vilkuttaminen. Hahmon täytyi myös kyetä siirtymään näiden kahden animaation välillä. Lopulta päädyttiin animaatiorutiiniin, jossa jokaisen ruudunpäivityksen alussa hahmo palautetaan oletustilaan eli paikalla seisomiseen, jonka jälkeen jokaista ruumiinosaa liikutellaan hienovaraisesti hengittävän hahmon vaikutelman luomiseksi. Tämän jälkeen tarkastetaan, onko “animState”-nimisellä muuttujalla johonkin tiettyyn animaatioon viittaavaa arvoa, jossa tapauksessa hahmon ruumiinosille tehdään vielä kyseiseen animaatioon liittyvät siirtelyt.

Hahmo siis suorittaa käytännössä kahta animaatiota päällekkäin. Esimerkiksi vilkuttamisen tapauksessa hahmo suorittaa ensin paikallaan seisomiseen liittyvät toiminnot, joiden jälkeen se suorittaa vielä vilkuttamiseen liittyvät toiminnot. Ratkaisu johtuu siitä, että tällä tavoin eri animaatioita kohti tarvitsee erikseen määritellä ainoastaan olennaisimmat toiminnot. Jos hahmo siis vilkuttaessaan on määrätty vain heiluttamaan oikeaa kättään edestakaisin, se jatkaa silti paikallaan seisomiseen liittyvän “hengittämisen” suorittamista. Hahmosta saadaan näin hieman elävämmän näköinen myös yksinkertaisten animaatioiden aikana.

Animaatiorutiini pitää myös kirjaa nykyisessä animaatiotilassa vietetyistä ruudunpäivityksistä “ticksInState”-muuttujalla, jonka ylittäessä tietyn arvon hahmo lopettaa sen hetkisen animaationsa ja siirtyy takaisin paikallaan seisontaan.

Alla olevassa koodissa on prototyypin jokaisella ruudunpäivityksellä ajettava animaatiorutiini. Koodia on tiivistetty siten, että tässä ovat esillä vain hahmon oikean käden liikkeisiin ja näytölle ilmestyvään puhekuplaan liittyvät animaatiofunktiot.

```
function handleTick(event) {

 boneArmRightUpper.rotation = 0;
 boneArmRightLower.rotation = 0;
 moro.visible = false;

 boneArmRightUpper.rotation -=
 Math.sin(createjs.Ticker.getTicks() / 16) * rotSpeed * 2;
 boneArmRightLower.rotation -=
 Math.cos(createjs.Ticker.getTicks() / 16) * rotSpeed * 2;

 if (animState == "wave") {
  boneArmRightUpper.rotation = -60;
```

```
 boneArmRightUpper.rotation +=
 Math.cos(createjs.Ticker.getTicks()/8) * rotSpeed * 20;

 boneArmRightLower.rotation = -100;

 boneArmRightLower.rotation +=
 Math.cos(createjs.Ticker.getTicks()/8) * rotSpeed * 20;

 moro.visible = true;
}

ticksInState += 1;

if(ticksInState>125) {
 animState = "idle";
 ticksInState = 0;
}

stage.update();
}
```

4.5 Prototyypeistä kohti valmista sovellusta

4.5.1 Lauseenkäsittelyn ja animaation yhdistäminen

Kun lauseenkäsittelyn ja animaatiorutiinien pohjatyö oli saatu valmiiksi, ne täytyi yhdistää samaan sovellukseen. Lauseenkäsittely ja hahmoanimaatio toimivat enimmäkseen omina itsenäisinä moduuleinaan, joten niiden yhdistäminen oli melko vaivatonta. Ainoa moduulien välinen yhteys on se, että hahmon on kyettävä vaihtamaan animaatioitaan samalla, kun tekoäly vastaa käyttäjän antamaan syötteeseen.

Kerrattakoon vielä, että animaatiorutiinissa tarkistetaan jokaisen ruudunpäivityksen aikana “animState”-muuttujan arvo ja siirrellään hahmon ruumiinosia sen mukaisesti. Hahmon kulloinkin suorittama animaatio siis riippuu yksinomaan tämän muuttujan arvosta. Näin ollen eri vastauksien yhteen voidaan liittää animaatioita lisäämällä niihin “_function_”-kontrollikoodi, joka kutsuu “animState”-muuttujan arvoa vaihtavaa funktiota. Esimerkiksi vastaus “I am happy! _function_setAnim_arg_happy” kutsuu samalla funktiota “setAnim(“happy”);”.

Lauseenkäsittelyn tuottama vastaus voitiin myös piirtää canvas-elementin sisälle EaselJS:n text-luokan avulla. Text-olioita käytetään tekstin piirtämiseen canvasille, ja ne toimivat kuten muutkin EaselJS:n näyttöobjektit. Näin ollen myös tekstiin oli mahdollista lisätä animaatiota, ja uudet vastaukset päädyttiin häivyttämään nopeasti sisään niiden ilmestyessä näytölle. Jos tekoäly esimerkiksi päätyisi antamaan täsmälleen saman vastauksen kahdesti peräkkäin, tästä ilmenisi että sovellus on kuitenkin vastaanottanut ja käsitellyt käyttäjän syötteen.

4.5.2 NestBotin antamien vastausten suunnittelu

Luvussa 4.3 kerrottiin, kuinka NestBot valitsee vastauksensa ja kykenee sisällyttämään siihen osan käyttäjän antamasta syötteestä. On kuitenkin huomioitava, että NestBot ei kykene hahmottamaan syötteestä kieliopillisia rakenteita tai semanttisia merkityksiä, vaan se reagoi ennalta ohjelmoiduilla tavoilla tiettyihin avainfraaseihin.

Huonosti suunnitellut vastaukset voi näin ollen johtaa tilanteisiin, jossa vastaus ei sovi enää kieliopillisesti tai semanttisesti yhteen käyttäjän antaman syötteen kanssa. Jos käyttäjän syöte esimerkiksi olisi “I am a great man”, ja NestBotin vastaus fraasiin “I am” olisi muodossa “I know many people that are _post_”, vastauksesta tulisi kieliopillisesti virheellinen “I know many people that are a great man”. NestBot ei kykene ymmärtämään, että “a great man” pitäisi tällaisessa lauseessa vaihtaa monikkoon (“great men”), eikä sen lauseenkäsittelyssä myöskään ole keinoja tällaisiin toimenpiteisiin. Varmempaa olisi siis vaihtaa tämä vastaus esimerkiksi muotoon “I heard that you _post_”, jolloin NestBot osaisi olemassa olevilla lauseenkäsittelytoiminnoillaan muuttaa käyttäjän syötteen toiseen persoonaan ja antaa vastauksekseen “I heard that you are a great man.”

Kaikkiin tilanteisiin ei tietenkään pystytä varautumaan. Käyttäjä saattaa itsekin käyttää virheellistä kielioppia, jolloin NestBotilla on hyvin huonot mahdollisuudet tuottaa syötteen pohjalta oikeaoppisesti muotoiltuja lauseita. Käytännössä NestBotia jouduttiin testaamaan mahdollisimman monella erilaisella, paikoin järjettömälläkin syötteellä, ja korjaamaan esiintyneet ongelmatilanteet.

Vastausten priorisointi osoittautui myös tarkkuutta vaativaksi työksi. Jos “random-Response”-muuttuja on epätosi, NestBotin antama vastaus riippuu puhtaasti vastausten prioriteetista, eli käytännössä siitä järjestyksestä, jossa ne on määritetty. Alla esitettävät esimerkit pätevät siis vain siinä tapauksessa, että muuttuja on epätosi ja NestBotin vastaus valitaan ensimmäisenä määritetyn avainfraasin perusteella.

Hyvänä ohjenuorana vastausten priorisoinnissa toimi se, että useammista sanoista koostuvat ja merkitykseltään tarkka-alaisemmat avainfraasit ovat prioriteetiltaan suurempia kuin lyhyemmät ja yleiskäyttöisemmät fraasit. Taulukossa 5 on esimerkki siitä, miten erilaisia avainfraaseja voitaisiin tällaisen periaatteen mukaan priorisoida.

Taulukko 5. Esimerkki avainfraasien priorisoinnista.

Avainfraasi	**Prioriteetti**	**Perustelu**
“coffee”	Korkea	Viittaa suoraan yksittäiseen käsitteeseen - käyttäjä haluaa selvästikin puhua kahvista, joten vastauksenkin tulisi käsitellä kahvia
“I like”	Keskitaso	Ei viittaa mihinkään tiettyyn käsitteeseen, mutta ilmaisee selvästi jotain tekemistä - käyttäjä pitää jostakin
“I”	Pieni	Ei viittaa suoraan mihinkään käsitteeseen tai tekemiseen - käyttäjä tekee ehkä jotain, mutta ei voida tietää mitä

Jos NestBotin avainfraasiluettelo siis koostuisi kokonaisuudessaan taulukon 5 kolmesta avainfraasista, syöte “I like coffee” saisi NestBotin antamaan korkeimman prioriteetin omaavaan “coffee”-avainfraasiin liittyvän vastauksen. Syötteestä voidaan erottaa sekä avainfraasit “I like” ja “I”, mutta “coffee” on prioriteetiltaan korkein ja vastaus määrittyy näin ollen sen pohjalta. “I like cheese” puolestaan saisi NestBotin antamaan “I like” -fraasiin liittyvän vastauksen, kun taas “I eat cheese” sisältää enää vain avainfraasin “I”.

Avainfraasien huono priorisointi saattaa näin ollen johtaa tilanteisiin, joissa NestBotille on loogisesti mahdotonta valita syötteestä tiettyjä fraaseja ensimmäiseksi avainfraasiksi. Jos taulukossa 5 fraasi “I” olisikin priorisoitu korkeammalle kuin fraasi “I

like”, NestBot ei käytännössä ikinä antaisi “I like” -fraasiin liittyvää vastausta. Tällaisessa tapauksessa “I like” itsessään sisältäisi aina korkeammalle priorisoidun avainfraasin, jota ohjelma suosisi joka kerralla.

4.5.3 Ilmaisukykyisempi hahmo eleillä ja ilmeillä

Hahmografiikan ja animaation ohjelmallisen toteutuksen valmistuttua olikin aika ryhtyä luomaan virtuaalihahmolle lopullista graafista ilmettä. Animaatioprototyyppiä varten luodun palikkahahmon ruumiinosien paikalle piirrettiin kokonaan uudet grafiikat. Työn tuloksena syntyi charmikas pukumies Jeff, joka on näkyvillä kuviossa 8.

Kuvio 8. Virtuaalihahmo Jeff.

Animaatioprototyypissä hahmo saatiin vilkuttamaan tarkistamalla jokaisella ruudunpäivityksellä ajettavan animaatiorutiinin aikana, vastasiko ”animState”-muuttujan arvo tiettyä merkkijonoa. Uusia eleitä varten animaatiorutiiniin lisättiin uusia animaatiotiloja ehtolauseineen ja asiaankuuluvine ruumiinosien liikkeineen.

Kun hahmo oli saatu suorittamaan erilaisia eleitä vastausten antamisen yhteydessä, kävi kuitenkin ilmi, että pelkkä ruumiinosien heiluttaminen ei vielä luonut hahmosta kovin eläväistä vaikutelmaa. Animaatiojärjestelmää täytyi laajentaa vielä siten, että hahmo kykenisi tekemään erilaisia ilmeitä.

Käytännössä eri ilmeet ovat “vaihtopäitä” hahmolle. Hahmon ilmeen vaihtuessa hahmon päätä edustava bittikartta korvataan siis sellaisella bittikartalla, jossa on kuva samasta päästä erilaisella ilmeellä. Lopullisen sovelluksen hahmografiikkaa tuottaessa täytyi siis ottaa huomioon, miten monta erilaista ilmettä hahmon on kyettävä tekemään. Lopulta päädyttiin viiteen ilmeeseen, jotka näkyvät kuviossa 9: hahmon oletustila eli tavallinen hymy, ymmärtäväisempi hymy käyttäjän rohkaisuun ongelmatilanteissa, surullinen ja anova katse, nyrpeä irvistys sekä riemukas nauru (vasemmalta oikealle).

Kuvio 9. Virtuaalihahmo Jeffin eri ilmeet.

4.5.4 Hahmografiikan kokoaminen yhteen bittikarttaan

Ilmeiden piirtämisen jälkeen ilmeni kuitenkin, että hahmon grafiikka koostui jo melko monesta tiedostosta. Viiden eri pään ja muiden ruumiinosien myötä hahmo oli jaettu yhteentoista erilliseen tiedostoon. Mikäli hahmografiikan muokkaamiselle tai kokonaan uudenlaisen hahmon piirtämiselle ilmenisi tarvetta, yhdentoista eri tiedoston käsittely sen aikana olisi jo turhan vaivalloista. Web-sovelluksissa on myös tärkeää pitää palvelimelle tehtävien HTTP-pyyntöjen määrä pienenä, joten kaikki hahmon grafiikka oli saatava sisällytettyä yhteen tiedostoon.

EaselJS:n SpriteSheet-luokka osoittautui tähän sopivaksi työkaluksi. Se mahdollistaa yhden kuvan jakamisen useampaan osa-alueeseen, jotta niitä voidaan käyttää yksittäisen objektin erillisinä animaatioruutuina. NestBotin tapauksessa sillä voidaan kuitenkin myös hakea useaan erilliseen objektiin grafiikka yksittäisestä bittikartasta. Kuviossa 10 (sivu 50) on näkyvillä kaikki hahmon grafiikka yhteen bittikarttaan sijoiteltuna. Kuvion punaiset alueet merkkaavat niitä kohtia, joilta bittikartta jaetaan osiksi.

Kuvio 10. SpriteSheet-oliota varten koottu bittikartta.

SpriteSheet vaatii toimiakseen sekä osiksi jaettavan bittikartan että täsmälliset määritykset siitä, mistä kohdista bittikartta pilkotaan ja kuinka moneen ruutuun. Säännöllisen kokoisten ja tasavälein sijoiteltujen ruutujen tapauksessa määrityksiksi riittää pelkkä ruutujen korkeus ja leveys. NestBotin hahmografiikka koostuu kuitenkin useista epäsäännöllisen kokoisista osista, joten jako vaatii melko paljon erilaisia määrityksiä. Alla olevassa koodissa on esimerkiksi määritelty hahmon viiden eri pään sijainnit bittikartassa.

```
var frameData = [];
frameData.push([164,7,128,121,0,64,116]);
frameData.push([17,7,128,121,0,64,116]);
frameData.push([17,128,128,121,0,64,116]);
frameData.push([17,249,128,121,0,64,116]);
frameData.push([17,370,128,121,0,64,116]);
```

Hakasulkeissa olevat numerot viittaavat tässä ruudun vasemman ylälaidan x-koordinaattiin, y-koordinaattiin, leveyteen, korkeuteen, käytettävän kuvan järjestyslukuun (tässä tapauksessa aina 0, sillä käytössä on ainoastaan yksi kuva), sekä ruudun akselin x- ja y-koordinaatit. Kun ruumiinosien koordinaatit bittikartassa oli määritelty, lopullinen SpriteSheet-olio saatiin luotua bittikartan ja koordinaattien pohjalta.

Prototyypissä yksittäisten ruumiinosien grafiikka määrättiin viittaamalla suoraan ruumiinosan bittikarttaan. SpriteSheetia käytettäessä jokaisen ruumiinosan on oltava siihen viittaava Sprite-olio. Sprite on näyttöobjekti, joka näyttää kerralla yhtä SpriteSheet-oliossa määriteltyä ruutua. Kutakin ruumiinosaa vastaavalle Spritelle määritettiin oikea kuva gotoAndStop-funktiolla, joka vaihtaa Spriten näyttämää ruutua. Esimerkiksi alla olevassa koodissa luodaan hahmon ruumiinosille omat Spritet ja määritetään niille oikeat grafiikat. GotoAndStop-funktiolle annettava numeroarvo viittaa tässä siihen järjestykseen, jossa SpriteSheetin ruudut on aiemmin määritelty.

```
var spriteData = {
  images: [img_spritesheet],
  frames: frameData
};
var characterSprite = new cre-
atejs.SpriteSheet(spriteData);

var head = new createjs.Sprite(characterSprite);
head.gotoAndStop(0);
var torso = new createjs.Sprite(characterSprite);
torso.gotoAndStop(5);
var armLeftUpper = new createjs.Sprite(characterSprite);
armLeftUpper.gotoAndStop(6);
var armLeftLower = new createjs.Sprite(characterSprite);
armLeftLower.gotoAndStop(7);
var armRightUpper = new createjs.Sprite(characterSprite);
armRightUpper.gotoAndStop(8);
var armRightLower = new createjs.Sprite(characterSprite);
armRightLower.gotoAndStop(9);
var legs = new createjs.Sprite(characterSprite);
legs.gotoAndStop(10);
```

Näiden muutosten jälkeen hahmografiikka toimi kuten aikaisemminkin, mutta se saatiin nopeammin latautuvaksi ja helpommin muokattavaksi. Grafiikan muokkausmahdollisuudet herättivät myös ajatuksia siitä, kuinka niistä saataisiin vielä hieman monipuolisempia. NestBotille ryhdyttiin näin ollen rakentamaan tukea vaihdettaville hahmografiikoille, eli “skineille”.

4.5.5 NestBotin skinnaus

Kertauksena NestBotin hahmografiikka koostuu kolmesta osasta: grafiikan sisältävästä bittikartasta, SpriteSheetin määrityksistä sekä luurangosta joka sisältää ruumiinosien hierarkkisen järjestyksen ja niiden sijainnit suhteessa toisiinsa. Nämä tiedot

päädyttiin laittamaan omaan JavaScript-tiedostoonsa, joka ladataan ennen itse sovelluksen käynnistämistä. Näin NestBotin käyttämä skini määräytyy yksinkertaisesti viittaamalla yhteen tiedostoon, joka sisältää kaikki skinin tarvitsemat ainutlaatuiset tiedot.

Bittikarttan tiedostosijainti tallennetaan aluksi omaan muuttujaansa, josta se sovelluksen käynnistyttyä välitetään resurssien esilataukseen.

SpriteSheetin paloittelu määritellään samaan tapaan kuin aiemminkin kuvailtiin, eli määrittämällä frameData-taulukkoon kunkin osan koordinaatit. Nämä määritykset ovat sisällytetty skinin tietoihin sitä varten, että esimerkiksi eri kokoisten ruumiinosien käyttäminen eri skineissä olisi mahdollista.

Ruumiinosien hierarkkiseen järjestykseen ei voida vaikuttaa skinin määrityksistä, mutta yksittäisille ruumiinosille on mahdollista määrittää omat x- ja y-koordinaattinsa. Tätä varten luotiin boneOffset-luokka, joka sisältää muuttujat näille koordinaateille ja funktiot niiden palauttamista varten.

```
function boneOffset(xOff, yOff) {
  this.xOffset = xOff;
  this.yOffset = yOff;

  this.getX = function() {
    return this.xOffset;
  }
  this.getY = function() {
    return this.yOffset;
  }
}
```

Kullekin ruumiinosalle voidaan siis määrittää omat koordinaattinsa suhteessa siihen ruumiinosaan, joka on hierarkkisessa järjestyksessä sen vanhempana.

```
var legsOffset = new boneOffset(0,36);
var armLeftUpperOffset = new boneOffset(-23,-33);
var armLeftLowerOffset = new boneOffset(-8,26);
var armRightUpperOffset = new boneOffset(23,-33);
var armRightLowerOffset = new boneOffset(8,26);
var headOffset = new boneOffset(0,-46);
```

Kun sovelluksen käynnistyttyä hahmon luurankoa rakennetaan, ruumiinosien koordinaatit haetaan boneOffset-luokan funktioilla.

```
var boneLegs = new createjs.Container();
boneLegs.addChild(legs);
boneLegs.x = legsOffset.getX();
boneLegs.y = legsOffset.getY();
```

Ruumiinosille määritellään vielä erilliset pyöritysarvonsa, jotta hahmon asentoa voidaan muokata. Tämä mahdollistaa hieman erilaisen ruumiinkielen omaavien hahmojen luomisen. Hahmo voi esimerkiksi pitää jatkuvasti päätään hieman kallellaan tai käsiään leveämmässä asennossa. Hahmon ruumiinosille annetaan aina animaatiorutiinin alussa tässä määritellyt pyöritysarvot, joten ne vaikuttavat kaikkiin hahmon tekemiin liikeisiin.

```
var legsRot = 0;
var armLeftUpperRot = 15;
var armLeftLowerRot = -10;
var armRightUpperRot = -15;
var armRightLowerRot = 10;
var headRot = 0;
```

Skinijärjestelmän testaamiseksi luotiin vaihtoehtoinen hahmografiikka, jolla on hieman erilaiset mittasuhteet ja elekieli. Tästä syntyi animetyttö Waifu, jonka pohjana toimiva bittikartta ja lopullinen sovellusgrafiikka näkyvät kuviossa 11 (sivu 54) Jeffin rinnalla. SpriteSheetin pohjana toimiva bittikartta on rakenteeltaan kummallakin hahmolla melkein samanlainen, mutta lopullisessa sovelluksessa Waifu on Jeffiin verrattuna kapeaharteisempi ja seisoo erilaisessa asennossa.

Kuvio 11. Kaksi erilaista skiniä.

4.6 Sovelluksen viimeistely

4.6.1 Tekstisyötteen käytettävyyden parantaminen JQuerylla

Lauseenkäsittelyn prototyyppiä luotaessa sovelluksen sisältävään HTML-dokumenttiin lisättiin tekstikenttä, johon käyttäjä voi kirjoittaa syötteensä. Tekstikentän vierelle lisättiin myös painike, jota painamalla sovellus vuorostaan tulkitsee kenttään kirjoitetun syötteen. Kuviossa 12 on kuvankaappaus sovelluksesta käyttöliittymän kanssa.

Kuvio 12. Sovellus sekä käyttäjäsyötteelle varattu tekstikenttä.

Sovelluksen käyttäminen ei ollut näin erityisen miellyttävää, sillä käyttäjä joutui jatkuvasti vuorottelemaan näppäimistön ja hiiren käytön välillä. Niinpä sovelluksessa otettiin vielä käyttöön JQuery, joka mahdollisti näppäimistön enter-näppäimen yhdistämisen suoraan samaan funktioon kuin painikkeen klikkaamisenkin. Funktio välittää tekstikentän sisällön lauseenkäsittelyllä ja tyhjentää samalla kentän seuraavaa syötettä varten. Alla oleva koodi sisältää näin ollen kaikki käyttöliittymää koskevat toiminnot.

```
$("#textInput").keyup(function(event){
  if(event.keyCode == 13){
    $("#textInputBtn").click();
  }
});

$("#textInputBtn").click(function(event) {
  handleString($("#textInput").val());
  $("#textInput").val("");
});
```

Sovellusta oli näin mahdollista käyttää kokonaan näppäimistön välityksellä. Klikattava painike jätettiin kuitenkin näkyviin, jotta sovelluksen käyttäminen olisi mahdollista myös mobiililaitteilla.

Tekstikentän HTML-elementille tehtiin vielä kaksi käyttökokemusta parantavaa asetusta. Sen autocomplete-parametri otettiin pois päältä, jotta selain ei yrittäisi automaattisesti täydentää tekstikenttää siihen aikaisemmin kirjoitetuilla syötteillä. Autofocus-parametri taas laitettiin päälle, jotta käyttäjä voi välittömästi sivun latauduttua kirjoittaa syötteen valitsematta tekstikenttää erikseen hiirellä. Sovelluksen käyttöliittymä oli siis lopulta määritelty HTML-koodissa seuraavanlaisesti:

```
<input id="textInput" type="text" autocomplete="off"
 autofocus="on”>
<input id="textInputBtn" type="submit" value="Submit">
```

4.6.2 NestBotin kolme osaa – hahmografiikka, sanasto ja sovellus

Luvussa 4.5.5 mainittiin jo, että kaikki yksittäiseen hahmoskiniin liittyvät määritykset tallennettiin omaan JavaScript-tiedostoonsa. Lopulta myös NestBotin sanasto pää-

dyttiin eristämään omaan tiedostoonsa, jotta se olisi mahdollisimman helposti muokattavissa. Tiedostossa määritellään kaikki ne taulukot, joista NestBotin sanasto koostuu: avainfraasit, synonyymit, ja korvattavat sanat.

Itse sovellus jää siis kokonaan omaan tiedostoonsa, joka sisältää vain lauseenkäsittelyn sekä sovelluksen yleiset grafiikkarutiinit. Grafiikkarutiinit koostuvat käynnistyksen aikana ajettavista toiminnoista, joissa määritellään canvas-elementin sisältö ja luodaan hahmon vartalolle hierarkkinen luuranko, sekä jokaisella ruudunpäivityksellä ajettavista animaatioista.

Animaatiorutiiniin sisältyvät myös erilaiset animaatiot, joita hahmo voi tehdä. Nämä olisi voitu eristää vielä oliomaisesti omaksi luokakseen, jonka oliot määriteltäisiin esimerkiksi hahmon skinin yhteydessä. Toisaalta rajallinen valikoima lujakoodattuja animaatioita takaa paremmin erilaisten skinien yhteensopivuuden. Näin ei siis päädytä esimerkiksi tilanteeseen, jossa sanastosta poimittu vastaus yrittäisi käynnistää animaatiota, jota käytössä olevassa skinissä ei olisi määritelty.

Sovelluksen käyttämä canvas-elementti ja syötteen mahdollistavat käyttöliittymäelementit sijaitsevat HTML-dokumentissa, jossa määritellään myös sovellukseen ja sen käyttämiin kirjastoihin kuuluvien JavaScript-tiedostojen sijainnit. HTML-dokumentin body-lohko on siis rakenteeltaan seuraavanlainen:

```
<body>
  <div id="nestbot">
    <canvas id="nestBotCanvas" width="700"
     height="360"/></canvas>
    <br/>
    <input id="textInput" type="text"
     autocomplete="off" autofocus="on”>
    <input id="textInputBtn" type="submit" value="Submit">
  </div>

  <script src="js/preloadjs-0.4.1.min.js"></script>
  <script src="js/easeljs-0.7.1.min.js"></script>
  <script src="js/jquery-1.11.0.min.js"></script>
  <script src="js/vocab.js"></script>
  <script src="js/skin-jeff.js"></script>
  <script src="js/nestbot.js"></script>
</body>
```

Aluksi siis ladataan sovelluksen käyttämät kirjastot: PreloadJS resurssien esilatausta, EaselJS grafiikkaa ja JQuery käyttöliittymää varten. Seuraavaksi ladataan NestBotin käyttämä sanasto ja hahmoskini, jotka alustavat tarpeelliset muuttujat lopuksi ladattavan sovelluksen käynnistämiseen.

4.6.3 Sanasto ja vastaukset

Kattavan sanaston kokoaminen NestBotille ei ollut opinnäytetyön tärkein tavoite, vaan päämääränä oli pikemminkin rakentaa sanastolle hyvä pohja, jota pystyttäisiin myöhemmin laajentamaan tarpeen mukaan. Näin ollen NestBotin sanaston kehitys painottui yleisiin fraaseihin ja kieliopillisiin rakenteisiin, eikä opinnäytetyön yhteydessä ryhdytty enää arvioimaan, millaiset asiasanat olisivat sovelluksen käyttöympäristön kannalta olennaisimpia.

Parhaaksi tavaksi sanaston laajentamiseen osoittautui NestBotin kanssa keskusteleminen. Kun NestBotille annettiin syötettä ikään kuin oikealle ihmiselle keskusteltaessa, saatiin hyvin selville, minkälaisiin avainfraaseihin sille pitäisi lisätä vastauksia tai mitkä sen valmiista vastauksista eivät sovikaan keskustelutilanteeseen. Lopullisen sovelluksen sanaston muokkaaminen oli nopea toimenpide, joten NestBotin keskustelutaitoja saatiin näin parannettua tehokkaasti eräänlaisessa palautesilmukassa.

Vastausten suunnittelussa noudatettiin osin ELIZAsta tuttuja periaatteita, eli NestBot saattaa pyytää tarkennuksia, vaihtaa puheenaihetta tai muuten puhua aiheen ympäri sellaisissa tilanteissa, jossa se ei onnistu tulkitsemaan käyttäjän syötettä tai kieliopillisesti oikean vastauksen antaminen olisi liian hankalaa. ”Ihanko totta”- tai ”kerro ihmeessä lisää”-tyyppiset vastaukset sopivat teoriassa mihin tahansa tilanteeseen, joskin NestBot saattoi joissakin tilanteissa päätyä antamaan liikaakin tällaisia vastauksia.

NestBotin vastauksille annettiin hieman persoonallisuutta ottamalla niihin mallia tieteistarinoiden yli-innokkaista ja pirteistä robottiapureista. Vastaukset sisältävät näin ollen paljon kohteliasta kielenkäyttöä, imelyyksiä ja käyttäjän kutsumista ystäväksi. Sanastoon lisättiin myös ns. pääsiäismuniksi joitakin yllättäviä avainfraaseja ja

vastauksia, kuten esimerkiksi NestBotin syvä loukkaantuminen sitä robotiksi nimitettäessä.

4.6.4 Puuttuvat ominaisuudet

Alun perin NestBotin oli tarkoitus olla integroitu osa FreeNest-alustaa, joka olisi mahdollistanut esimerkiksi työn alla olevan projektin tilanteen kysymisen NestBotilta suoraan. Lopulta tällaiset ominaisuudet eivät kuitenkaan toteutuneet, vaan opinnäytetyön yhteydessä luotu tekoäly on käytännössä itsenäinen web-sovellus, jolla ei ole minkäänlaisia yhteyksiä itsensä ulkopuolisiin tietovarastoihin.

Opinnäytetyön loppuvaiheessa kävi selväksi, ettei tällaisia ominaisuuksia enää saataisi toteutettua saatavilla olevilla aikaresursseilla. Aikaa käytettiin sen sijaan valmiiksi saadun koodin dokumentoimiseen ja kommentointiin, jotta sovellusta voitaisiin myöhemmin laajentaa mahdollisimman vaivattomasti. Sovelluksen koodi kommentoitiin jokaisen toiminnallisuuden kannalta olennaisen funktion kohdalta. Sovellukseen sisällytetyt hahmoskinit ja sen käyttämä sanasto kommentoitiin erityisen huolellisesti, ja ne sisältävät jo yksityiskohtaisia ohjeita omien muutosten tai laajennusten tekemiseen.

5 Pohdinta

5.1 Tavoitteet ja menetelmät

5.1.1 Tekoälyohjelmointi ja lauseenkäsittely

Opinnäytetyön pääasiallisena pyrkimyksenä oli tuottaa toimiva tekoäly, jonka kanssa on mahdollista käydä ainakin jollain tasolla järkeviä keskusteluita. Tehtävä oli kunnianhimoinen, joten aihetta pyrittiin lähestymään mahdollisimman monesta näkökulmasta.

Tekoälytutkimuksen historiaan ja menetelmiin perehtyminen paljasti, että tekoälyn keskeiset haasteet sivuavat usein monia tieteenaloja, eikä niille välttämättä ole olemassakaan mitään yksiselitteistä ratkaisua. Esimerkiksi lauseentulkinta on hyvin monimutkainen ja useita pulmia kattava tehtävä. Opinnäytetyölle oli siis asetettava jotkin selkeät rajat ottaen huomioon, että tavoitteena oli kuitenkin luoda laajempi yhtenäinen tekoälysovellus eikä esimerkiksi itsenäistä lauseenkäsittelyalgoritmia.

Aihepiiriin tutustuminen antoi opinnäytetyötä varten hyvän tietopohjan, jota vasten verrata sille asetettuja tavoitteita ja käytettäviä menetelmiä. Sovelluksen kehityksen kannalta oli tärkeää tietää, millaisia tekoälyjä oli jo olemassa. Niiden menetelmien ja ominaisuuksien tunteminen auttoi määrittelemään tarkemmin, miten opinnäytetyön tekoäly olisi järkevintä toteuttaa. Erityisen hyödyllistä oli ihmisen ja koneen vuorovaikutusta koskeva tutkimus, joka antoi sovelluksen kehitykseen muitakin kuin puhtaasti tiedonkäsittelyyn liittyviä näkökulmia.

Sosiaalisen vuorovaikutuksen huomioiminen auttoi lähestymään opinnäytetyössä ilmenneitä pulmia lateraalisemmin. Chatterbottien kehityksen historiasta löytyikin hyviä ennakkotapauksia, joissa monimutkaisia ongelmia oli ikään kuin ohitettu tiedostamalla ohjelman rajoitteet ja kiinnittämällä käyttäjän huomion pois niistä. Tavallisessa sovelluskehityksessä tällainen lähestymistapa voi toki olla tuhoisa. Keskustelevalla tekoälyllä on kuitenkin niin vähän käytännön sovelluksia, että sitä voitaisiin vielä

pitää eräänlaisena leluna, jossa vain käyttäjälle näkyvällä lopputuloksella on merkitystä.

5.1.2 Grafiikka ja animaatio

Chatterbotit ovat normaalisti puhtaasti tekstipohjaisia, joten grafiikan lisääminen antoi lähtökohtaisesti sovellukselle oman luonteensa. Grafiikan suhteen ensisijaisena tavoitteena oli antaa tekoälylle jonkinlainen persoona, jonka kanssa keskusteleminen tuntuisi hauskemmalta kuin kliinisen tekstirivin kanssa kommunikointi.

JavaScriptin grafiikkakirjastoihin tutustuminen kuului myös opinnäytetyöhön. Tarkoituksena oli löytää sellainen grafiikkakirjasto, joka sopisi erityisen hyvin hahmoanimaatioon ja joka olisi myös suoritusteholtaan mahdollisimman hyvä. Erilaisten kirjastojen kokeilu ja vertaileminen toisiinsa toimi itse asiassa hyvänä harjoitteluna lopullisen sovelluksen grafiikan ja animaatioiden luontiin. Esimerkiksi sulavien liikeratojen laskeminen sineillä ja kosineilla oli oivallus, joka syntyi grafiikkakirjastojen testaamisen lomassa.

Hahmografiikan suurin haaste oli luurankopohjaisen animaatiojärjestelmän toteuttaminen. Näytöllä näkyvän hahmo jaettiin hierarkkisesti erillisiin ruumiinosiin, joiden liikkeet vaikuttavat toisiinsa. Näin pyrittiin minimoimaan sekä käsin piirrettävän grafiikan että tarvittavien animaatiomääritysten määrä, kun hahmon jokaiselle liikkeelle ei tarvita erillistä animaatioruutua.

Grafiikkakirjastojen testauksessa käytteenotettu sinejä ja kosineja hyödyntävä animaatiomenetelmän auttoi pitämään hahmolle määritetyt animaatiot koodimäärältään suhteellisen pieninä. Valmiiksi määriteltyihin ruutuihin perustuvat perinteiset animaatiomenetelmät ohitettiin käytännössä kokonaan, ja valmiin sovelluksen hahmoanimaatiot perustuvat pikemminkin valmiiksi määriteltyihin algoritmeihin joiden mukaan hahmo liikkuu.

5.2 Aikaansaannokset

5.2.1 Persoonallinen tekoäly

Sovelluksen lauseenkäsittely oli lopulta yksinkertaisempi kuin esimerkiksi 1960-luvulla ohjelmoidussa ELIZA-tekoälyssä. Tarkoituksena ei lopulta ollutkaan rakentaa erityisen sofistikoitunutta tekoälyä, vaan yhdistellä olemassa olevia tekniikoita uutta käyttötarkoitusta varten. Tekoäly oli alusta asti tarkoitettu FreeNest-alustan markkinointityökaluksi, joten päämääränä oli luoda näillä tekniikoilla mahdollisimman näyttävä ja persoonallinen sovellus.

Tekoäly itsessään ei sisällä minkäänlaisen persoonallisuuden määritelmän ohjelmallista edustusta. Koodista itsestään siis ei löydy mitään, mikä suoraan viittaisi tekoälyhahmon luonteenpiirteisiin tai sen kokemiin tunteisiin. Näistä luotiin kuitenkin vaikutelma kiinnittämällä erityistä huomiota tekoälyn sanastoon ja sen antamiin vastauksiin. NestBotille luotiin omanlainen puhetapansa, joka vastasi tietynlaista persoonallisuutta.

Grafiikan lisääminen antoi tekoälylle myös huomattavasti enemmän persoonallisuutta. Erilaisten ilmeiden ja eleiden lisääminen vastausten yhteyteen toi erilaisia tunnetiloja esille paremmin kuin pelkkä tekstirivi. Tällaiset tunne-ilmaisut auttoivat luomaan vaikutelmaa tekoälystä pelkkää lauseenkäsittelyalgoritmia kokonaisvaltaisempana hahmona.

5.2.2 Muokattava hahmografiikka

Persoonallisen toteutuksen jälkeen merkittävin saavutus grafiikassa oli, että sen ohjelmallisesta toteutuksesta saatiin suhteellisen modulaarinen ja helposti muokattava. Opinnäytetyön kirjoitushetkellä FreeNest-alustalla ei esimerkiksi ollut minkäänlaista virallista markkinoinnissa käytettävää maskottihahmoa, mutta sellaisen mahdollisesti syntyessä se saataisiin melko pienellä vaivannäöllä käyttöön myös NestBotille.

Sovelluksen mukaan paketoitu hahmografiikka toimii myös hyvänä käytännön esimerkkinä sen graafisista ominaisuuksista. Kaksi erilaista hahmoa havainnollistavat vaihdettavien hahmoskinien toimintaa ja kenties sitäkin, kuinka hahmon ulkonäkö voi vaikuttaa tulkintoihin tekoälyn persoonallisuudesta.

5.2.3 Yhtenäinen alusta tekoälylle ja hahmografiikalle

Opinnäytetyön toimeksiannossa oli alun perin kyse ”projektieläimestä”, eli nimenomaan käyttäjän kanssa keskustelevasta virtuaalihahmosta. Tällaisen hahmon toteuttamiseen vaaditaan sekä toimivaa tekoälyä että jonkinlaista hahmografiikkaa. Näin ollen ei voida vähätellä sen merkitystä, että toisistaan eroavat osa-alueet saatiin toimimaan keskenään yhtenäisessä sovelluksessa.

Vastaavanlaiset ratkaisut eivät ole aivan tyypillisiä, sillä tekoälytutkimus ja hahmoanimaatio eivät sinänsä liity toisiinsa millään tavalla. Opinnäytetyön sovellusta voitaisiinkin pitää alustana, jossa tekoälyn lauseenkäsittely on integroitu ruudulla liikkuvaan animaatiohahmoon. Sovelluksen visuaalisen esityksen merkittävimmät tavoitteet on siis jo saavutettu, ja sillä on hyvä pohja mahdolliselle jatkokehitykselle.

5.3 Jatkokehitys

5.3.1 NestBotin soveltuvuus jatkokehitykseen

Opinnäytetyön suurin saavuttamattomaksi jäänyt tavoite oli sovelluksen integroiminen FreeNest-alustaan. Lauseenkäsittelyn ohjelmointiin kului odottamattoman paljon aikaresursseja, eikä alustaintegraation toteuttaminen ollut opinnäytetyölle rajatuissa puitteissa enää mahdollista. Korvaavaksi tavoitteeksi otettiin sen sijaan koodin kommentointi ja jäsentely sellaiseksi, että sovellus voidaan myöhemmin tuoda mahdollisimman vaivattomasti lähemmäksi alkuperäistä visiota.

Suhteellisen monimutkainen luurankopohjainen hahmoanimaatio saatiin NestBotissa eristettyä omiksi moduuleikseen siten, että hahmografiikan muokkaaminen tai luo-

minen ei vaadi merkittävää teknistä osaamista tai NestBotin ohjelmalogiikan tuntemusta.

NestBotin lauseenkäsittelystä tuli rajoitteistaan huolimatta toimiva järjestelmä, joka mahdollisti jo järkevän oloisten keskustelujen käymisen tekoälyn kanssa. Sen sanastosta tuli ennen kaikkea helposti ja nopeasti muokattava, ja siinä käytetyt suunnitteluperiaatteet on dokumentoitu sekä opinnäytetyössä että koodin kommenttiriveillä. Sanaston ja vastausten laajentaminen tai täysimittainen uudistaminenkaan ei näin ollen tulisi vaatimaan kohtuutonta työmäärään. NestBotille ei juurikaan annettu esimerkiksi projektinhallintaan liittyvää sanastoa, mutta sellaisen lisääminen olisi nopeaa ja vaivatonta.

Lauseenkäsittelyn oheisfunktiot mahdollistavat myös ulkoisten funktioiden ajamisen tiettyjen vastausten yhteydessä. Opinnäytetyössä tätä ominaisuutta ehdittiin käyttää ainoastaan hahmon animaatioiden välillä siirtymiseen, mutta käytännössä se mahdollistaisi myös tiedon noutamisen FreeNest-alustasta syöttämällä tekoälyllä tiettyjä avainfraaseja. Sovellukseen jo sisältyvä JQuery-kirjasto mahdollistaisi palvelimen puoleisen koodin ajamisen JavaScriptistä kutsuttuna.

5.3.2 Uudet ominaisuudet

Muokattavuus ja jatkokehityksen mahdollistaminen olivat tärkeitä tavoitteita NestBotin kehityksessä, ja yleisesti ottaen ohjelman eri osa-alueista saatiinkin melko helposti laajennettavia. Hahmografiikkaan liittyvät animaatiot toimivat kuitenkin hieman omanlaisella logiikallaan. Niiden koodista ei välttämättä ilmene välittömästi, millaisia liikkeitä kukin sini- tai kosini-pohjainen liikerata tuottaa näytölle piirrettävällä hahmolla. Jos hahmon animaatioiden muokkaukselle tai laajentamiselle ilmenisi tarvetta, voitaisiin harkita erillisen animaatioeditorin tuottamista, jossa liikeradat voitaisiin määritellä canvas-elementtiin liitetyillä käyttöliittymäelementeillä ja tehdyt muutokset nähtäisiin reaaliajassa. Valmiiseen animaatioon liittyvä koodi voitaisiin esimerkiksi kopioida leikepöydälle, josta se voitaisiin liittää sovelluksen animaatiorutiiniin.

Merkittävimmät uudet ominaisuudet liittyisivät sovelluksen ja FreeNest-alustan väliseen kommunikaatioon. FreeNest itsessään koostuu enimmäkseen useasta erillisestä avoimen lähdekoodin sovelluksesta, joten esimerkiksi projektin tilanteen esittäminen NestBotin välityksellä jouduttaisiin todennäköisesti toteuttamaan aina sovelluskohtaisesti sen mukaan, mistä alustan osasta tieto haetaan.

Lähteet

Hofstadter, D. 1995. Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought. New York: BasicBooks.

Hutchens, J. 1997. How to Pass the Turing Test by Cheating. Viitattu 11.4.2014. http://www.nyu.edu/gsas/dept/philo/courses/mindsandmachines/Papers/hutchens96how.pdf

Inhimillinen kone, konemainen ihminen. 2001. Toim. E. Hyvönen. Helsinki: Yliopistopaino.

Landsteiner, N. 2005. Elizabot.js. Viitattu 18.4.2014. http://www.masswerk.at/elizabot/

Nurmi J-E., Ahonen T., Lyytinen H., Lyytinen P., Pulkkinen L., Ruoppila I. 2006. Ihmisen psykologinen kehitys. WSOY Oppimateriaalit Oy, Helsinki. 1. painos.

Smith, E., Nolen-Hoeksema, S., Fredrickson, B., Loftus, G. 2003. Atkinson & Hilgard's Introduction to Psychology. 14. p. Belmont, CA: Wadsworth/Thomson Learning.

Viitaniemi, V. 2008. Osaavatko koneet ajatella? Tekoäly saapuu osaksi modernia yhteiskuntaa. Helsinki: Books on Demand GmbH.